UNIVERSIDAD POLITECNICA DE MADRID

Facultad de Informática

Dpto. de Lenguajes y Sistemas Informáticos
e Ingeniería de Software

Tesis Doctoral

DERIVACION DEDUCTIVA DE PROGRAMAS FUNCIONALES CON PATRONES

Autor: **Jesús Angel Velázquez Iturbide**

Director: **José Antonio Cerrada Somolinos**

Noviembre de 1990

A mis padres

DERIVACION DEDUCTIVA DE PROGRAMAS FUNCIONALES CON PATRONES

RESUMEN

Una de las dificultades principales en el desarrollo de software es la ausencia de un marco conceptual adecuado para su estudio. Una propuesta la constituye el modelo transformativo, que entiende el desarrollo de software como un proceso iterativo de transformación de especificaciones: se parte de una especificación inicial que va transformándose sucesivamente hasta obtener una especificación final que se toma como programa. Este modelo básico puede llevarse a la práctica de varias maneras. En concreto, la aproximación deductiva toma una sentencia lógica como especificación inicial y su proceso transformador consiste en la demostración de la sentencia; como producto secundario de la demostración se deriva un programa que satisface la especificación inicial.

La tesis desarrolla un método deductivo para la derivación de programas funcionales con patrones, escritos en un lenguaje similar a Hope. El método utiliza una lógica multigénero, cuya relación con el lenguaje de programación es estudiada. También se identifican los esquemas de demostración necesarios para la derivación de funciones con patrones, basados en la demostración independiente de varias subsentencias. Cada subsentencia proporciona una subespecificación de una ecuación del futuro programa a derivar.

Nuestro método deductivo está inspirado en uno previo de Zohar Manna y Richard Waldinger, conocido como el cuadro deductivo, que deriva programas en un lenguaje similar a Lisp. El nuevo método es una modificación del cuadro de estos autores, que incorpora géneros y permite demostrar una especificación mediante varios cuadros. Cada cuadro demuestra una subespecificación y por tanto deriva una ecuación del programa. Se prevén mecanismos para que los programas derivados puedan contener definiciones locales con patrones y variables anónimas y sinónimas y para que las funciones auxiliares derivadas no usen variables de las funciones principales.

La tesis se completa con varios ejemplos de aplicación, un mecanismo que independentiza el método del lenguaje de programación y un prototipo de entorno interactivo de derivación deductiva.

Categorías y descriptores de materia CR

D.1.1 [**Técnicas de programación**]: Programación funcional; D.2.10 [**Ingeniería de software**]: Diseño - métodos; F.3.1 [**Lógica y significado de los programas**]: Especificación, verificación y razonamiento sobre programas - lógica de programas; F.3.3 [**Lógica y significado de los programas**]: Estudios de construcciones de programas - construcciones funcionales; esquemas de programa y de recursión; I.2.2 [**Inteligencia artificial**]: Programación automática - síntesis de programas; I.2.3 [**Inteligencia artificial**]: Deducción y demostración de teoremas]: extracción de respuesta/razón; inducción matemática.

Términos generales

Programación funcional, síntesis de programas, demostración de teoremas.

Otras palabras claves y expresiones

Funciones con patrones, cuadro deductivo, especificación parcial, inducción estructural, teorema de descomposición.

DEDUCTIVE DERIVATION OF FUNCTIONAL PROGRAMS WITH PATTERNS

ABSTRACT

One of the main difficulties in software development is the lack of an adequate conceptual framework of study. The transformational model is one such proposal that conceives software development as an iterative process of specifications transformation: an initial specification is developed and successively transformed until a final specification is obtained and taken as a program. This basic model can be implemented in several ways. The deductive approach takes a logical sentence as the initial specification and its proof constitutes the transformational process; as a byproduct of the proof, a program which satisfies the initial specification is derived.

In the thesis, a deductive method for the derivation of Hope-like functional programs with patterns is developed. The method uses a many-sorted logic, whose relation to the programming language is studied. Also the proof schemes necessary for the derivation of functional programs with patterns, based on the independent proof of several subsentences, are identified. Each subsentence provides a subspecification of one equation of the future program to be derived.

Our deductive method is inspired on a previous one by Zohar Manna and Richard Waldinger, known as the deductive tableau, which derives Lisp-like programs. The new method incorporates sorts in the tableau and allows to prove a sentence with several tableaux. Each tableau proves a subspecification and therefore derives an equation of the program. Mechanisms are included to allow the derived programs to contain local definitions with patterns and anonymous and synonymous variables; also, the derived auxiliary functions cannot reference parameters of their main functions.

The thesis is completed with several application examples, a mechanism to make the method independent from the programming language and an interactive environment prototype for deductive derivation.

CR categories and subject descriptors
D.1.1 [**Programming techniques**]: Functional programming; D.2.10 [**Software engineering**]: Design - methodologies; F.3.1 [**Logics and meanings of programs**]: Specifying and verifying and reasoning about programs - logics of programs; F.3.3 [**Logics and meanings of programs**]: Studies of program constructs - functional constructs; program and recursion schemes; I.2.2 [**Artificial intelligence**]: Automatic programming - program synthesis; I.2.3 [**Artificial intelligence**]: Deduction and theorem proving - answer/reason extraction; mathematical induction.

General terms
Functional programming, program synthesis, theorem proving.

Additional key words and phrases
Functions with patterns, deductive tableau, structural induction, partial specification, descomposition theorem.

AGRADECIMIENTOS

Quiero expresar mi profundo agradecimiento hacia las siguientes personas en relación con esta tesis:

José Antonio Cerrada, como Director de la tesis, ha confiado en mí para la realización de la tesis, apoyándome y aconsejándome.

Manuel Collado, como Director del Departamento de Lenguajes y Sistemas Informáticos, me ha permitido elegir el tema de la tesis y me ha dado facilidades para su realización.

Los profesores y becarios del Grupo Docente de Programación, sobre todo Angel Sánchez, me han apoyado moralmente y me han dado su amistad.

Los miembros de la Biblioteca de la Facultad de Informática, especialmente Begoña Méndez y Paloma Crego, con su excelente trabajo han hecho posible que accediera a la mayor parte del material de estudio seleccionado.

Juan Pavón me ha proporcionado material de gran utilidad.

Todos aquéllos que con sus escritos han contribuido a aumentar mi conocimiento de lógica y programación, sobre todo Zohar Manna y Richard Waldinger.

INDICE

1. INTRODUCCION

Pretendemos con este primer capítulo exponer al lector los objetivos perseguidos al realizar la tesis y proporcionarle una pequeña guía para la lectura de la misma. Para ello hemos organizado el capítulo en cuatro partes. La primera parte presenta el campo de la programación transformativa de una manera general; se describe el modelo de desarrollo de software por transformaciones y los conceptos básicos de transformación de programas. A continuación se exponen de una manera superficial los sistemas transformativos que más relación tienen con la tesis. Estos dos primeros puntos proporcionan el contexto necesario para explicar razonadamente los objetivos de la tesis, que se exponen en el apartado tercero. Por último, se describe la estructura de la tesis para facilitar la lectura de la misma.

1.1. NOCIONES BASICAS DE TRANSFORMACION DE PROGRAMAS

Puede decirse que en el mundo occidental toda actividad productiva no artística pasa por una serie de etapas resumibles en tres [ACM90]: artesanía, práctica comercial e ingeniería profesional. Dado que la realización de programas es una actividad productiva y que hay razones técnicas y sociales para no considerarla artística, podemos integrarla en el proceso evolutivo anterior. Pronto observamos que actualmente la producción de software se encuadra más fácilmente en las dos primeras etapas que en la tercera y más deseada. Por supuesto, el software es un término tan amplio que esta afirmación puede resultar demasiado categórica: hay campos o aspectos concretos donde se ha alcanzado sobradamente el estado final (p.ej.

generación de código de bajo nivel o ciertas aplicaciones de gestión). Sin embargo, el estado general de la producción de software aún adolece de graves carencias.

Desde la aparición de la informática se viene intentando mejorar el proceso productivo de software, con logros importantes, como arquitecturas más potentes, lenguajes de alto nivel y modelos y entornos de desarrollo de software. Sin embargo, ninguno de ellos ha constituido la "bala de plata" que mate al hombre-lobo que es la producción de software [Brooks87].

Una de las principales corrientes de investigación se caracteriza por intentar crear una base matemática rigurosa sobre la que asentar el software. El objetivo es desarrollar instrumentos de análisis y síntesis similares al cálculo en las ingenierías. Estos instrumentos serían la base de la enseñanza [DCGMTTY88] y la práctica de la informática. La formalización de los diversos aspectos del desarrollo de software permitiría su realización automática, o al menos automatizada mediante entornos. De hecho hay aspectos ya formalizados y ampliamente utilizados de una manera automática [GoMo87] (p.ej. analizadores sintácticos, comprobadores de tipos y controladores de versiones). Sin embargo, estos logros son pequeños en relación con lo que aún queda por comprender.

Podemos concebir una situación ideal en la que el usuario explica sus necesidades en lenguaje natural, sea cual sea el dominio de aplicación, y ésta es generada automáticamente. Por supuesto, esta situación no es realista hoy [RiWa88], y puede que no lo sea nunca. Sin embargo, podríamos acercarnos a este ideal si se consiguiere comprender y formalizar el software y usar el ordenador como ayuda. La dificultad principal está en establecer un marco conceptual adecuado para el estudio del software [ACM90], sobre todo de las etapas iniciales: requisitos, especificación y diseño. Su dificultad de formalización proviene de que son los aspectos más creativos, con un gran

abanico de opciones en su realización.

Las ideas anteriores guían a un modelo de desarrollo de software alternativo de los tradicionales (p.ej. en cascada o en espiral), llamado modelo transformacional o transformativo. El modelo tiene un gran valor como marco conceptual donde estudiar el desarrollo de software, por lo que al menos puede adoptarse para el simple estudio del software. Existen múltiples propuestas del modelo transformativo, que difieren en aspectos menores. Para concretar el discurso, en lo sucesivo nos referimos a la propuesta de [BaChGr83], que reproducimos en la figura 1.1. El lector interesado puede consultar [Agresti86, Pepper84, Velázquez90] para exposiciones más extensas del tema.

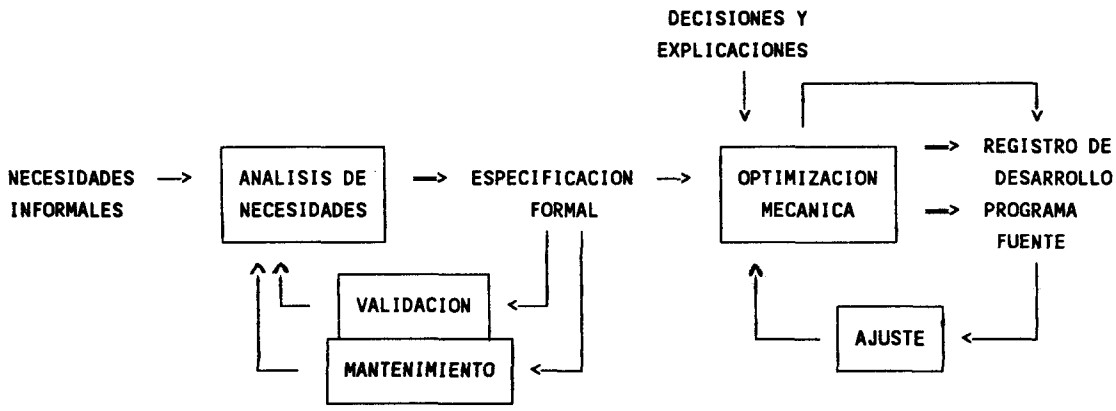


Figura 1.1. Modelo de la programación transformativa

El modelo transformativo parte de las necesidades del usuario, normalmente expuestas de una manera informal, ambigua e incompleta. El análisis de estas necesidades sirve para desarrollar una especificación formal, que debe recoger las necesidades de una manera precisa y comprensible por el usuario. Entonces se va transformando la especificación paso a paso hasta obtener un programa correcto y eficiente. Así pues, la implementación es un proceso iterativo de transformación de

programas, realizado mediante ordenador. Las transformaciones son decisiones de diseño de diversa naturaleza (p.ej. selección de un algoritmo o mezcla de bucles) y que son lógicamente correctas, con lo que se garantiza que el programa desarrollado cumple la especificación. Durante el proceso de implementación se genera automáticamente un registro de las transformaciones realizadas, así como las decisiones de diseño implicadas, llamado desarrollo formal o registro de desarrollo. Por último, el mantenimiento se realiza sobre la especificación, ya que es donde se encuentra la función del programa, reimplementándose posteriormente según el mismo proceso de transformación.

Obsérvese que la especificación es el objeto central del modelo, ya que sobre él se realizan la validación con las necesidades del usuario, las transformaciones y el mantenimiento del software. Al final del proceso nos encontramos con tres productos: la especificación, el programa desarrollado y el desarrollo formal.

Podemos apreciar mejor las ventajas del modelo transformativo comparándolo con el modelo de software convencional [BaChGr83]:

- La especificación es formal y sobre ella el usuario valida sus necesidades; en el modelo clásico la especificación es informal, validándose el código ya desarrollado. De este modo pueden detectarse inconsistencias o faltas al comienzo del proceso. Si además el lenguaje de especificación es ejecutable, se dispone directamente de un prototipo que facilita la validación de la especificación y que está totalmente integrado en el proceso.
- El sistema realiza la gestión de los aspectos mecánicos del desarrollo, como conservar la consistencia de los detalles del programa. En consecuencia, el programador sólo necesita concentrarse en los aspectos creativos del desarrollo, y por tanto puede obtener programas más efi-

cientes.

- Se evita la etapa de prueba sobre el programa final ya que el producto final es correcto, dada la corrección formal de las transformaciones y la falta de errores en su aplicación por el ordenador.
- El mantenimiento es más fácil de realizar debido a dos razones:
  - El mantenimiento se efectúa sobre la especificación, lo cual es más sencillo de hacer que sobre el código, que incluye decisiones de diseño ajenas a la funcionalidad del sistema. Esto podría permitir incluso su realización por el usuario.
  - El proceso transformativo genera automáticamente documentación sobre el diseño en forma de un registro de desarrollo, evitando la pérdida de las decisiones de diseño. Esta información puede ser utilizada en la reimplementación (incluso automática) de los aspectos no modificados de la especificación.

No hay una forma única de realizar este esquema básico de desarrollo. Básicamente se trata de llevar el principio de refino progresivo [Wirth73] hasta sus últimas consecuencias. El proceso consiste en partir de una especificación E_0 y realizar una secuencia de pasos de refino hasta conseguir un programa final **P**. Obviamente la dificultad de realizar un paso es mucho menor que la de realizar el programa completo. Esquemáticamente [Sannella88]:

$$E_0 \rightarrow \ldots \rightarrow E_i \rightarrow E_{i+1} \rightarrow \ldots \rightarrow E_n^* = P$$

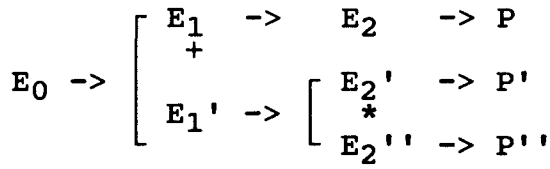
Técnicamente el refino de una especificación E_i en otra E_{i+1} se realiza mediante la aplicación a E_i de una regla de transformación, que produce como resultado E_{i+1}. Una regla de transformación representa una relación semántica entre dos esquemas de especificación. La relación está "dirigida", en el sentido de que la aplicación de la regla produce una especificación más cercana al programa final.

Una regla de transformación R_j consta de un patrón inicial PI_j, un patrón final PF_j y una condición de habilitación C_j. La aplicación de una regla de transformación R_j a una especificación E_i implica hacer coincidir el patrón inicial PI_j de la regla, mediante una instanciación θ de sus variables, con una parte o subespecificación SE_i de E_i. Posteriormente se evalúa la condición instanciada $C\theta$ y, si se cumple, se sustituye $SE_i = (PI_j)\theta$ en E_i por la instanciación $(PF_j)\theta$, dando lugar a E_{i+1}. Es decir, si representamos E_i como $E_i[(PI_j)\theta]$, E_{i+1} es $E_i[(PF_j)\theta]$ (ver notación en el Apéndice A).

Una transformación es correcta si existe una relación semántica entre los esquemas, generalmente de equivalencia. En todo caso debe tener propiedades de orden y de monotonicidad [PaSt83]. Las propiedades de orden, como reflexividad y transitividad, son necesarias para poder componer transformaciones. Las propiedades de monotonicidad garantizan que cambios locales correctos mantienen la corrección global.

Las reglas de transformación pueden ser de diversa naturaleza. Según su ámbito de aplicación pueden dividirse en reglas locales y globales. Las primeras sirven para relacionar construcciones sintácticas del lenguaje, describir propiedades de construcciones sintácticas y expresar conocimiento del dominio. Las segundas sirven para propósitos más generales, como comprobaciones de consistencia o representación de técnicas de programación. Entre medias hay una gama gradual de posibilidades.

Si la especificación es grande, puede hacerse difícil de manejar. Una solución es permitir su descomposición en subespecificaciones, que pueden refinarse por separado. Un ejemplo de desarrollo [Sannella88] con dos descomposiciones y seis pasos de refino se recoge en el siguiente diagrama:



En el diagrama, + y * representan operaciones de composición de especificaciones. Por tanto, se habría obtenido el programa **P+(P'*P'')** a partir de la especificación E_0. Si dichos operadores no son utilizables al nivel de programa, deben utilizarse otros correspondientes a éstos en dicho nivel. Las propiedades de monotonicidad de las transformaciones son especialmente importantes para garantizar la corrección de la composición de especificaciones.

Puede observarse que el método transformativo tiene varias consecuencias adicionales:

- Debe representarse explícitamente, en las reglas de transformación, conocimiento de programación.
- Si hay varios refinos realizables en una especificación dada, se explicita la existencia de varios diseños alternativos.

El conjunto total de alternativas de refino existentes en el desarrollo de un programa producen un árbol de búsqueda. El proceso de desarrollo de un programa concreto se puede ver como el recorrido de un camino desde la raíz del árbol hasta la hoja que representa el programa desarrollado. Un subárbol contiene una familia de programas que comparten decisiones de diseño comunes (las que forman el camino que va desde la raíz del árbol hasta la raíz del subárbol). El estudio de estos árboles por personas es irrealizable si no son programas obvios, mientras que un entorno puede facilitarlo.

El árbol de búsqueda mencionado se refiere a diseños factibles, es decir, que conducen a algún programa final. Esto, sin embargo, no significa que durante el proceso de desarrollo no existan alternativas correspondientes a malas decisiones de

diseño, y por tanto infactibles. Lo normal será que el proceso de desarrollo sea tentativo, incluyendo el retroceso como una parte del mismo. Podemos ampliar el árbol de búsqueda incorporando todos aquellos caminos de desarrollo inviables, obteniendo una visión más realista del conjunto de alternativas de refino existentes durante el desarrollo.

El lector puede intuir la existencia de una gran variedad de sistemas transformativos al concretar los elementos del modelo básico expuesto y de su utilización práctica. Podemos clasificar los métodos de transformación según varios criterios [PaSt83, Sannella88]. Veamos algunas de sus características:

- **Actividad.** Tal como se ha planteado el modelo transformativo, su actividad es el desarrollo completo de programas eficientes. También puede utilizarse para objetivos más modestos, como la sola síntesis u optimización de programas. Por síntesis se entiende el desarrollo de un programa, quizá ineficiente, a partir de una especificación no algorítmica, mientras que una optimización es la mejora, bajo algún criterio, de un programa ya existente. Por último, el modelo puede utilizarse como marco conceptual para el estudio del software (sobre todo de la lógica de los programas), es decir, con fines investigadores y docentes pero no productivos.

  Hay algunas actividades que coinciden parcialmente con el modelo transformativo. Así el diseño de lenguajes de altísimo nivel y sus "supercompiladores" (ver artículos referidos en [Feather87] o contenidos en [BiGu83, RiWa86]) comparte las ideas básicas de la transformación de programas, pero es menos ambicioso. La adaptación de programas a lenguajes o entornos distintos también se enfoca a veces análogamente al modelo transformativo [Krieg84]. Otro campo relacionado es la construcción de ayudantes de programación y tutores inteligentes (ver [RiWa90] y las colecciones de artículos de [RiWa86,

[SlBr82]). Por último encontramos trabajos en inteligencia artificial que intentan imitar el comportamiento humano en la síntesis de planes y programas [CoFe82,cap.XV, Sussman75].

- **Dominio de aplicación.** Normalmente el dominio de aplicación no está restringido a priori, aunque en la práctica sólo pueden desarrollarse programas sobre aquellos dominios de los que se ha suministrado conocimiento al sistema. También suele estar limitado el tamaño de los programas manejables.

- **Lenguajes.** Deben identificarse los lenguajes de especificación y de programación y la relación entre ambos. También hay otras cuestiones, no menos importantes, tales como de qué forma se produce la transición entre el lenguaje de especificación y el de programación, qué significa y cuándo es correcto el refino de una especificación en otra y si es posible el refino de programas. Obsérvese que la actividad del sistema está relacionada con los lenguajes, ya que si el lenguaje de especificación y el de programación son el mismo, siempre se realiza optimización de programas.

  La diferencia entre lenguajes de especificación y de programación no siempre es fácil de establecer. A veces se utiliza un único lenguaje "de amplio espectro", que puede usarse para escribir cualquier producto intermedio entre una especificación y un programa eficiente, ambos incluidos. En caso de diferenciar entre el lenguaje de especificación y de programación, tenemos un amplio surtido donde elegir. Los lenguajes de especificación varían de lenguaje natural restringido a lenguajes con distinto grado de naturaleza declarativa y operativa. Los lenguajes de programación suelen catalogarse de la manera usual en lenguajes lógicos, funcionales, imperativos de alto nivel, etc.

- **Reglas de transformación.** Hay tres cuestiones fundamentales sobre ellas: qué reglas utilizar, cómo establecer su corrección y cómo derivar nuevas reglas. Estas cuestiones están muy influidas por la organización de las reglas, bien como catálogo bien como conjunto generador. La primera es una colección estructurada de reglas de transformación, cada una relevante para un aspecto particular del proceso de desarrollo. Al no adoptar un enfoque homogéneo de la programación, la colección puede estar incompleta. La segunda organización consiste en un pequeño conjunto de transformaciones elementales, que recogen principios básicos de programación. Con frecuencia se necesita agruparlas para construir nuevas reglas. Suelen utilizarse en combinación con conocimiento adicional sobre el dominio de aplicación.

- **Ayudas por ordenador.** En caso de que el modelo no permanezca puramente teórico, sino que quiera utilizarse para el desarrollo de programas, deben crearse entornos específicos. Estos entornos se caracterizan por rasgos como grado de automatización del proceso, formas de manejo de las reglas, disponibilidad de información sobre el proceso de transformación y facilidad de documentación del proceso transformativo.

  El sistema puede ser automático, semiautomático o completamente controlado por el usuario. Por desgracia, el primer caso sólo se da en dominios muy concretos o imponiendo restricciones al lenguaje de especificación. En el segundo, el sistema puede realizar automáticamente ciertas tareas, pero en general es el usuario el responsable de la derivación. La automatización de ciertas partes rutinarias del desarrollo es esencial si se quiere eliminar complejidad al desarrollo transformativo. En su forma más básica el ahorro en gestión de consistencia del programa es superado con creces por el esfuerzo de selección de reglas, resultando más complejo y menos práctico que el desarrollo normal.

El conjunto de reglas de transformación suelen ir acompañadas de un mecanismo de gestión. Este facilita el acceso y selección de reglas. Una facilidad importante es la comprobación de condiciones en el programa, para permitir la aplicación de reglas. También se puede permitir la adición de reglas nuevas; a veces es el sistema el que genera nuevas reglas mediante operadores de composición y el que prueba su corrección. Asimismo, el sistema puede dar consejo al usuario de varias formas (p.ej. midiendo el efecto de una transformación o incluyendo información estratégica sobre la selección de reglas).

Uno de los aspectos más importantes del método transformativo es la posibilidad de registrar el proceso de derivación de un programa. La forma más sencilla de registro es, obviamente, una mera repetición de la sesión. Un mecanismo más elaborado debe estructurar el desarrollo según objetivos, estrategias y decisiones de diseño; sólo así puede utilizarse para reimplementar parte de una especificación modificada.

1.2. PANORAMICA DE ALGUNOS METODOS TRANSFORMATIVOS

La bibliografía sobre sistemas transformativos es bastante extensa. Podemos citar [BaFe82,cap.X, Biermann85, Fernández84,cap.1, Sannella88] y sobre todo [Feather87, PaSt83] como descripciones panorámicas de diversos sistemas transformativos. Además, el lector interesado puede encontrar recopilaciones de artículos en [Agresti86, BiGu83, BiGuKo84, Meertens87, Pepper84, RiWa86]. A continuación describimos muy brevemente algunos sistemas con un conjunto generativo de reglas que permiten desarrollar programas funcionales. Incluimos tres métodos de síntesis (mediante deducción lógica) y uno de optimización.

METODOS DEDUCTIVOS

Los métodos deductivos toman como especificación inicial una sentencia lógica que afirma la existencia de una función que satisface ciertas condiciones e intentan demostrar su validez lógica, obteniendo como producto de la demostración un programa funcional que satisface la especificación. Más concretamente, la especificación de una función **f** corresponde a cierta sentencia lógica:

$$(\forall\ x)\ (\exists\ z)\ \text{if } E[x] \text{ then } S[x,z]$$

que expresa que para cualesquiera valores de entrada **x** que satisfagan la condición de entrada **E**, puede obtenerse un valor de salida **z** que satisfaga la relación de entrada-salida **S**.

Los métodos deductivos se remontan a mediados de los años 60, época en que surgió un gran interés por el uso del ordenador para demostrar teoremas automáticamente. Una aplicación de estos sistemas sería desarrollar sistemas que contestaran a problemas resolubles por deducción. Una rama de este trabajo (p.ej. [Green69]) pretendía que el sistema proporcionara un programa como respuesta a una sentencia de especificación. Sin embargo, la dificultad de obtener programas recursivos o iterativos frenó temporalmente esta vía. Centrándonos en la derivación de programas funcionales, debemos esperar hasta [MaWa80] para obtener un sistema general, llamado el cuadro deductivo.

El **cuadro deductivo** de Zohar Manna y Richard Waldinger [MaWa80] se basa en experiencias previas con un sistema transformativo no puramente deductivo [MaWa79] y posteriormente ha evolucionado hacia el sistema deductivo descrito en [MaWa87]. Un cuadro deductivo es una tabla formada por un conjunto de filas y tres columnas. La columna de asertos incluye axiomas e hipótesis tomadas como ciertos (normalmente condi-

ciones de entrada o hipótesis de condición), la columna de objetivos contiene las sentencias que en cada momento quedan por demostrar y la columna del término de salida recoge el término funcional derivado en cada momento. Cada fila contiene un aserto o un objetivo y opcionalmente un término de salida.

El cuadro inicial (sin hipótesis de inducción) para una especificación genérica como la anterior es:

asertos	objetivos	f(a)
E[a]		
	S[a,z]	z

donde **a** representa un valor de entrada cualquiera y el cuadro incluye dos filas, la primera con la condición de entrada como aserto, y la segunda con la condición de entrada-salida como objetivo por demostrar y la variable **z** como "programa" obtenido hasta el momento.

El proceso deductivo consiste en aplicar sucesivamente alguna regla de deducción a filas existentes, añadiendo al cuadro el resultado de la deducción en forma de una o varias filas nuevas. El proceso deductivo termina cuando se obtiene un objetivo evidentemente cierto, es decir, la sentencia **true**. La regla principal es la regla de resolución [Robinson65, Murray82], pero se incluyen algunas otras que reducen la longitud de la demostración, p.ej. simplif..cación de sentencias lógicas o reemplazamiento de términos por igualdad. Las reglas pueden aplicarse a sentencias en cualquier formato, no necesariamente en alguna forma normal (p.ej. clausal), lo cual conserva la intuición sobre la deducción en curso.

Las nuevas filas tienen un término de salida cada vez más detallado, ya que van recogiendo información de la deducción parcial efectuada. Este término se modifica de una fila a la

siguiente al sustituir una variable por un término, como resultado de una unificación de subsentencias, o al introducir una expresión if-then-else formada por una subsentencia (la condición) y dos términos de salida ya existentes en el cuadro. Al final de la deducción, el término de salida contiene un programa completo.

El sistema tiene una estructura intuitiva de usar y unas reglas potentes y claramente definidas. El proceso deductivo permite tanto la síntesis como la optimización de programas funcionales. Además, se prevén mecanismos para definir funciones auxiliares. Como contrapartida, los lenguajes de especificación y programación están claramente diferenciados (lógica de primer orden y términos funcionales tipo Lisp), lo cual permite la derivación de cualquier programa funcional excepto predicados, ya que éstos pertenecen al lenguaje de especificación pero no al de programación. Otro inconveniente es que la información de tipos debe manejarse explícitamente mediante predicados, algo rutinario y tedioso de hacer.

El método se ha utilizado para la síntesis de una gran variedad de programas funcionales. Aparte de algunos programas sencillos [MaWa80, MaWa89], el método se ha aplicado a la derivación de un algoritmo de unificación [MaWa81, Nardi89], diversos algoritmos de ordenación de listas [Traugott89] y varios algoritmos numéricos basados en búsqueda binaria [MaWa87].

En el mismo año se publicó otro método deductivo [Bibel80] debido a Wolfgang Bibel. Igual que el cuadro deductivo entra en la tradición de demostración de teoremas por resolución, el **método heurístico** de Bibel se encuadra en la de deducción natural. Aun así, este autor no se compromete con ningún método formal concreto, sino que sus razonamientos los hace de una manera intuitiva.

En su forma más básica el método consiste en modificar la

sentencia de especificación mediante la aplicación de una secuencia de estrategias. Primero se conjeturan dos valores de salida más ciertas hipótesis que deben satisfacer éstos, conjeturas que se introducen en la especificación inicial. Dado que puede haber varias formas de crear las conjeturas, las posibles nuevas especificaciones se ordenan usando información del dominio sobre la facilidad de determinación de los valores y de satisfación de las hipótesis. Hecho esto se toma la sentencia de especificación modificada mínima y se reescribe usando una sencuencia de tres estrategias que respectivamente pretenden dejarla en forma normal disjuntiva, en forma recursiva y con los predicados de condición en forma evaluable por máquina. Para problemas más complicados se generalizan algunas de estas estrategias y se proporcionan otras adicionales. El programa final no está expresado en ningún lenguaje ni estilo de programación, sino que es una sentencia lógica susceptible de ser ejecutada (recursivamente) si se la impone un control.

A pesar de las afirmaciones optimistas de Bibel sobre la uniformidad y elegancia del método, en conjunto resulta un método farragoso y de dudosa generalidad, donde se mezcla la lógica de las reglas de transformación (deducción natural) y su control (estrategias). Además la pretendida uniformidad depende del uso de distintas bases de conocimiento (dominio de aplicación, esquemas de recursión) y de la efectividad de las estrategias, que no deja de ser una conjetura. El valor principal del método son algunas de las estrategias que incorpora y su orden de aplicación, que a pesar de su imprecisión, pueden ser útiles en un sistema deductivo más estructurado.

De una manera informal con el método se han desarrollado algoritmos muy diversos: encontrar el máximo elemento de una lista, un árbol de mínimo coste recubridor de un grafo y algunos otros de dificultad media. Existe un prototipo de sistema automático de síntesis [BiHö84].

Un tercer método deductivo, llamado de **reducción de pro-**

blemas [Smith83], deriva programas que se ajustan a ciertos esquemas. Un esquema de función es una definición de función con una estructura sintáctica fija pero con algunas partes sin concretar. Así, un esquema restringido de divide-y-vencerás es:

```
F(x) <= if condición(x)
        then ResolverDirectamente(x)
        else Componer°(FxF)°Descomponer(x) ;
```

donde **F**, **ResolverDirectamente**, **Componer** y **Descomponer** son metasímbolos que deben instanciarse con símbolos de funciones concretas,

condición es un metasímbolo que debe instanciarse con una expresión booleana,

° representa la composición de funciones, y

x representa el producto de funciones, definido como (FxG)(x,y) = (F(x),G(y)).

Dada una especificación inicial de una función **f**, usamos el esquema anterior para intentar obtener un programa de divide-y-vencerás que satisfaga la especificación. Basta instanciar adecuadamente los metasímbolos del esquema (en particular **F** por **f**). Un esquema de función también establece una relación entre la especificación de la función principal y las especificaciones de las funciones auxiliares. Por tanto, dada una especificación y un esquema de función, pueden deducirse las especificaciones de las funciones auxiliares. El modo concreto de obtener las subespecificaciones a partir de la especificación constituye una estrategia de diseño asociada al esquema. Smith utiliza un sistema lógico de deducción natural [Smith82] para deducir las subespecificaciones.

El método de reducción de problemas consta de dos fases para el desarrollo de programas, una descendente y otra ascendente. En la fase primera, dada una especificación inicial y un conjunto de esquemas y estrategias de diseño asociadas, se selecciona un esquema y una estrategia de diseño

y se obtienen ciertas especificaciones auxiliares. El proceso se aplica recursivamente a cada especificación auxiliar, terminando cada rama al encontrar una especificación directamente satisfactible por alguna función. El resultado del proceso descrito es un árbol donde cada nodo contiene una especificación y un esquema de programa. Los hijos de un nodo son las especificaciones de las funciones auxiliares utilizadas en el esquema asociado al nodo. Las hojas son las especificaciones elementales satisfactibles directamente.

La segunda fase del método de reducción de problemas crea los programas definitivos a partir del árbol desarrollado. Se comienza con las especificaciones elementales incluidas en las hojas del árbol y se les asocia la función que las satisface. Cada vez que un nodo tiene funciones asociadas con todas sus especificaciones hijas, se forma el programa que satisface la especificación del nodo instanciando el esquema del nodo con las funciones hijas. El proceso termina al contruir el programa que satisface la especificación de la raíz del árbol, es decir, la especificación inicial.

El método de reducción de problemas resulta adecuado para derivar funciones de cualquier complejidad porque sólo afecta al tamaño del árbol descompositor. El proceso proporciona un programa muy estructurado. También proporciona alguna ayuda para completar la condición de entrada de las especificaciones [Smith83]. El inconveniente principal del método es su falta de generalidad puesto que se restringe a esquemas prefijados. Esto también provoca que los programas finales estén excesivamente fragmentados.

El método se ha aplicado a programas condicionales no recursivos [Smith85c] y de divide-y-vencerás [Smith85a, Smith85b], estando automatizado. Se ha aplicado a diversos problemas, como ordenación de listas, hallar el producto cartesiano de dos conjuntos y encontrar el casco convexo de un conjunto de puntos planares.

METODOS DE OPTIMIZACION

Varios trabajos se realizaron a comienzos de los años 70 en el área de optimización de programas funcionales. Estos sistemas toman un programa funcional y por medio de transformaciones realizadas sobre el mismo programa, obtienen un programa equivalente pero más eficiente. Quizá el sistema más interesante sea el realizado por Burstall y Darlington [BuDa77], conocido como **método de despliegue-pliegue.**

El método de despliegue-pliegue precisa el uso de algún lenguaje funcional con una disciplina de tipos basada en constructores. En concreto se han utilizado NPL [Burstall77] y Hope [BuMaSa80]. Esta característica permite definir una función mediante varias ecuaciones de recurrencia, cada una con los parámetros formados por un conjunto de constructores distintos, es decir, cada una con un patrón distinto. Aparte de ventajas para el programador convencional, desde el punto de vista de la transformación de programas la separación en ecuaciones con patrones diferentes facilita la transformación de cada caso por separado.

La optimización de una función comienza con la definición de una función nueva no recursiva que se prevé que puede realizar el cálculo (u otro parecido) más eficientemente que la función inicial. La elección de esta función no suele ser evidente, llamándose con frecuencia "eureka" a este paso. Posteriormente se generan varias ecuaciones para dicha función mediante la instanciación de algunos parámetros a términos formados con constructores distintos. A continuación se transforma cada una de estas ecuaciones para obtener una formulación recursiva. Por último se redefine la función inicial en términos de la nueva, que por tanto queda como una función auxiliar.

El conjunto de reglas de transformación disponible para la

optimización es pequeño, recogiendo una serie de conceptos intuitivos de programación:

1. Definición. Se añade una ecuación no recursiva que define una función nueva.
2. Instanciación. Se añade una instancia, por sustitución de algunos parámetros, de una ecuación ya existente.
3. Despliegue. Dadas dos ecuaciones E<=E' y F<=F', donde el lado derecho F' de la segunda ecuación contiene alguna instancia Eθ de E, se reemplaza en F' dicha aparición de Eθ por E'θ.
4. Pliegue. Dadas dos ecuaciones E<=E' y F<=F', donde el lado derecho F' de la segunda ecuación contiene alguna instancia E'θ de E', se reemplaza en F' dicha aparición de E'θ por Eθ.
5. Abstracción. Se reescribe la parte derecha de una ecuación mediante la introducción de una definición local.
6. Leyes. Se reescribe el lado derecho de una ecuación aplicando alguna propiedad de función (p.ej. asociatividad).

Dentro del proceso anterior, la transformación de una ecuación (nueva) no recursiva en otra recursiva suele constar de una serie de despliegues seguidas de algunas leyes y abstracciones y unos pliegues finales. Esta es la razón de que el método se conozca con el nombre de despliegue-pliegue.

El sistema resultante es sencillo, pero permite realizar una gran variedad de optimizaciones y es fácilmente automatizable [Darlington81a]. Además pueden definirse nuevas transformaciones en función de las ya existentes [Darlington81b, Feather82], con lo que se mantiene la corrección del proceso y se aumenta la potencia del sistema. También tiene cierta capacidad de síntesis si el lenguaje se amplía con expresiones de conjuntos [Darlington75]. Como inconvenientes deben destacarse la falta de generalidad del sistema y que sólo garantiza una equivalencia débil de los programas derivados [Kott78] (léase

una broma de C.A.R. Hoare al respecto en [Meertens87], pág. 410).

El método de despliegue-pliegue se ha utilizado extensivamente, dada su sencillez conceptual y amplia aplicabilidad. También ha inspirado otros trabajos de transformación de programas, tanto aplicados a lenguajes lógicos [TaSa84] como funcionales [Scherlis81, Reddy88]. Debe destacarse (a pesar de tratar con lenguajes atipados, bajo la influencia de Manna y Waldinger) el trabajo de W. Scherlis porque las reglas que propone garantizan la equivalencia fuerte de los programas transformados.

1.3. OBJETIVOS DE LA TESIS

Veamos qué objetivos concretos de transformación de programas nos hemos propuesto. Para tal fin conviene comentar qué lugar ocupan los lenguajes de programación dentro del modelo. Es algo obvio que cuanto mayor es el nivel de abstracción de un lenguaje de programación, más fácil resulta expresar programas con él. Hay dos razones importantes de este hecho: una es que la mayor cercanía del nivel de abstracción del lenguaje al del dominio de aplicación permite expresar algoritmos y tipos de datos sin detalles adicionales, extraños al dominio; también sucede que los lenguajes de bajo nivel mezclan la lógica de la solución con el control de su ejecución, resultando en semánticas más oscuras.

Dentro del modelo transformativo es corriente [BMPP89] que la transformación de la especificación inicial se haga de una manera gradual, con unos programas iniciales sencillos pero ineficientes y unos programas finales menos claros pero más eficientes. Este proceso puede realizarse más fácilmente si los primeros programas se desarrollan en lenguajes declarativos, porque expresan algoritmos independientes de la arquitectura del computador; posteriormente pueden optimizarse, trans-

formándose en programas dependientes de la arquitectura (imperativos de alto nivel, paralelos, etc.). Dadas las cada vez más desarrolladas técnicas de implementación de lenguajes declarativos [Peyton87], incluso es posible que en el futuro el programa final esté expresado en un lenguaje declarativo. (Obsérvese que el proceso corriente de traducción de programas escritos en un lenguaje imperativo de alto nivel a otro imperativo de bajo nivel también entra en este proceso; sólo suele ignorarse porque se sabe hacer de una manera automática y eficiente.) Por tanto, parece razonable concentrarse en la obtención de programas declarativos a partir de especificaciones.

La restricción a los lenguajes declarativos tiene dos ventajas, como ya hemos mencionado antes. Primero, su obtención a partir de especificaciones resultará más fácil. Segundo, su semántica más sencilla, y en particular la propiedad de transparencia consultiva ("referential transparency") [EiSa87], facilita la manipulación directa de programas.

Este es el contexto en el que vamos a fijar los objetivos de la tesis, al que añadimos tres características adicionales. La primera consiste en elegir el estilo funcional frente al lógico. La opción se basa en que queremos obtener algoritmos, que siempre aparecen en un programa funcional, mientras que no es así necesariamente en programas lógicos. Además, la búsqueda deductiva que hay durante la ejecución de un programa lógico restringe su eficiencia, mientras que un programa funcional expresa un método directo y eficiente de cálculo del resultado deseado. En realidad sí tendremos cierta búsqueda con los programas funcionales, pero una sola vez: durante la derivación del programa.

La segunda característica es que de momento nos interesa más la comprensión de los procesos de síntesis que su automatización. Aunque estos dos aspectos están relacionados, los distinguimos de manera análoga a la separación entre lógica y

control en lenguajes declarativos. Por tanto, nos centramos en la lógica de la derivación de programas, dejando de momento su control.

Una última opción es que utilicemos un marco conceptual general. En concreto se debe poder obtener la mayor variedad posible de programas funcionales, tanto mediante síntesis como mediante optimización. Por supuesto, el lenguaje de programación debe ser un lenguaje funcional moderno.

Una vez definidas las características deseadas del trabajo resultante, vamos a concretar un marco conceptual que permita desarrollar dicho trabajo. Examinando los sistemas transformativos del apartado anterior, nos parece que el sistema de Manna y Waldinger [MaWa87] es el más satisfactorio. Este método permite realizar tanto la síntesis como la optimización de (prácticamente) cualquier programa funcional de primer orden (es decir, sin funciones de orden superior). Además su estructura conserva la intuición durante el proceso transformativo.

El sistema deductivo de Manna y Waldinger está pensado para producir programas atipados tipo Lisp. Por tanto, sólo puede aplicarse de una manera restringida a lenguajes más modernos. En concreto estos lenguajes tienen un conjunto de características (p.ej. tipos de datos y definiciones de función con patrones) de las que carece Lisp y que permiten un estilo de programación más claro. Así pues, dicho sistema sólo permitiría obtener un subconjunto del total de programas expresables con un lenguaje moderno. Por ejemplo, las funciones derivadas estarían expresadas mediante una única ecuación.

El objetivo de la tesis es adaptar el cuadro deductivo de Manna y Waldinger para la síntesis de programas funcionales basados en ecuaciones de recurrencia con patrones. Esto conlleva la posibilidad de conseguir algunas características adicionales, tales como evitar el uso de funciones selectoras de

tipos estructurados y utilizar definiciones locales con patrones. Dado que tenemos que elegir un lenguaje funcional adecuado para nuestros propósitos, adoptamos un lenguaje similar a Hope; de este modo se podrían realizar fácilmente comparaciones con el método de despliegue/pliegue.

1.4. ESTRUCTURA DE LA TESIS

Aparte de este primer capítulo introductorio, la tesis consta de 5 capítulos más. El Capítulo 2 contiene los principios lógicos necesarios para la derivación de funciones con patrones, pero sin comprometerse con ningún sistema deductivo concreto, lo cual se retrasa hasta el Capítulo 3. Describimos los lenguajes de programación y especificación utilizados y su relación. Esta relación es la base para justificar la transición de especificaciones a programas y la definición de la teoría lógica que usamos para la derivación de funciones. También se identifican varios esquemas de demostración de sentencias lógicas, que se utilizarán para la derivación de funciones con patrones.

El Capítulo 3 describe en detalle el sistema deductivo. Una buena parte del sistema es similar al de Manna y Waldinger, incluyendo su estructura y reglas de deducción, por lo que se expone brevemente. Los aspectos propios del uso de patrones ocupan una buena parte de la exposición: cómo obtener funciones con rango tupla, varias ecuaciones de recurrencia, definiciones locales con patrones y variables anónimas y sinónimas. También damos un procedimiento completo de derivación, haciendo hincapié en los aspectos, más problemáticos, de la introducción de llamadas recursivas y funciones auxiliares.

El nuevo cuadro deductivo se ilustra aplicándolo en el Capítulo 4 a varios problemas. La elección de los ejemplos se ha hecho buscando cierta variedad en sus características:

objetivo de la derivación (síntesis u optimización de funciones), dominio de aplicación (tipos de datos), forma de la derivación y posible uso de definiciones locales y parámetros anónimos y sinónimos. Podemos destacar una función que halla el máximo elemento de una lista y otra de ordenación de listas.

El Capítulo 5 contiene un conjunto de cuestiones subsidiarias al tema principal de la tesis. Se estudia la adaptación de ciertos esquemas inductivos al cuadro de Manna y Waldinger, la independencia del método respecto al lenguaje de programación funcional y el desarrollo de un prototipo de entorno interactivo para la derivación deductiva de funciones.

El capítulo final contiene las conclusiones obtenidas de nuestro trabajo y algunas propuestas de trabajo futuro.

Finalmente se incluyen dos apéndices que complementan la información de los capítulos descritos. El Apéndice A da una relación de vocablos y notaciones usados, pero no definidos, en el cuerpo de la tesis. El Apéndice B contiene la teoría lógica usada por el autor para la derivación de varios programas, incluyendo los descritos en el Capítulo 4.

2. PRINCIPIOS FUNCIONALES Y LOGICOS

Antes de desarrollar el sistema deductivo de derivación de funciones necesitamos ciertos principios que se elaboran en este capítulo: los lenguajes de especificación y programación utilizados más los principios lógicos necesarios para la derivación de funciones definidas mediante patrones. En los dos primeros apartados describimos respectivamente los lenguajes de especificación y de programación; el primero es el lenguaje de la lógica multigénero de predicados de primer orden, mientras que el segundo es un lenguaje funcional de primer orden similar al lenguaje funcional Hope. En el apartado tercero exponemos de una forma muy general la derivación de funciones a partir de especificaciones. Posteriormente analizamos la relación entre programas funcionales y las teorías lógicas de los tipos declarados. Por último, exponemos dos formas distintas de demostrar una sentencia: por inducción y por descomposición; el primer esquema permite derivar programas recursivos y el segundo, programas con patrones. Como caso especial, la inducción estructural combina características de ambos esquemas.

2.1. LENGUAJE DE PROGRAMACION

La programación funcional es un estilo de programación basado en el cálculo con funciones. No describimos en detalle dicho estilo, aunque los conceptos funcionales utilizados se encuentran resumidos en el apartado A.7. El lector puede encontrar buenas introducciones a la programación funcional en [FiHa88, Henderson80, MacLennan90, Reade89, SaVe88], así como a lenguajes de programación funcional en [AlSuSu85, Bailey90,

Bird88, WiHo84, Wikström87]. En nuestro caso utilizamos un lenguaje de programación similar al lenguaje Hope [BuMaSa80, Bailey90, FiHa88], sólo que basado en la lógica de primer orden en vez del cálculo lambda. (Por consiguiente no contiene características polimórficas ni de orden superior.) Por similitud lo llamamos miniHope. Aunque es descrito brevemente en el apartado A.7, a continuación incluimos las características predefinidas del lenguaje.

TIPOS DE DATOS Y FUNCIONES PREDEFINIDOS

El lenguaje contiene ciertos tipos de datos predefinidos. Estos tipos son los enteros no negativos, las listas de enteros y los valores de verdad, y sus valores posibles son obvios. En miniHope nombramos a estos tipos con los símbolos **entero**, **lista** y **booleano**. El uso del tipo booleano está restringido a predicados y conectivas. (Debe recordarse que el lenguaje es monomórfico, a diferencia de Hope. En consecuencia **lista** es un tipo, no un operador de tipo. Si se quiere disponer de listas de otros valores, debe declararse un nuevo tipo de listas.)

El lenguaje contiene el símbolo '#', que en Hope se utiliza como operador de tipo "producto cartesiano" y sirve para construir agrupaciones de valores ya existentes. Una agrupación de valores se llama tupla y puede tener una longitud arbitraria pero finita; las tuplas de longitud 2, 3, ..., n se llaman respectivamente pares, ternas, ..., n-tuplas. Los **n** valores que forman una n-tupla no tienen porqué ser del mismo tipo. En miniHope '#' no es un operador de tipo, sino un convenio sintáctico para representar distintos tipos. De manera estricta cada tipo tupla debería declararse explícitamente con una declaración de tipo. Sin embargo, seguimos el convenio de Hope por comodidad. Por ejemplo, **entero#lista** es el tipo de los pares entero-lista, que utilizamos en vez del símbolo de tipo más convencional **parenterolista**.

Frecuentemente hablaremos de tipos estructurados para referirnos a tipos cuyos valores contienen valores de otros tipos. Es el caso de tuplas, listas, árboles y conjuntos.

La expresión **t : T** significa que el término funcional **t** es de tipo **T**. De esta manera, algunos ejemplos de valores son:

```
1047 : entero
[0,1,2,3] : lista
true : booleano
(6, 8) : entero#entero
(6, [99, 23]) : entero#lista
```

Obsérvese que una lista de enteros se representa encerrando éstos entre corchetes y separándolos con comas. Asimismo, una tupla se representa encerrando sus elementos entre paréntesis y separados con una coma.

Todos los tipos se declaran mediante constructores, incluso los tipos predefinidos (aunque en la práctica estos tipos no necesiten declararse). Así, las declaraciones de los tipos entero y lista, en caso de necesitarse, serían:

```
data entero == 0 ++ succ entero ;
data lista == nil ++ entero::lista ;
```

Obsérvese que los enteros no negativos se forman a partir de cero y el sucesor de algún otro entero. Asimismo las listas se forman a partir de la lista vacía y de añadir un entero a una lista ya formada; el constructor **::** (pronúnciese "cons") es infijo. Sin embargo, para mayor comodidad no escribiremos los términos **succ succ succ 0** y **0::(succ 0::nil)**, sino los más usuales **3** y **[0,1]**. También consideramos que el constructor **::** es asociativo por la derecha, de forma que la expresión **1::2::3::nil** es igual que **1::(2::(3::nil))**. El símbolo de constructor **::** tiene mayor prioridad que el símbolo **<>** de

concatenación.

Las tuplas también se forman a partir de sus elementos constituyentes mediante el constructor coma ',', es decir, que **(1,7)** se forma aplicando el constructor coma a **1** y **7**. De manera estricta cada tipo tupla posible debería declararse con un constructor distinto. El uso de un único constructor coma es un convenio que sirve para usar más cómodamente las tuplas.

También existe un conjunto de funciones predefinidas sobre los tipos predefinidos. Todas las funciones, excepto los constructores, pueden definirse explícitamente, incluso las predefinidas. La razón de incluir funciones predefinidas es facilitar la tarea del programador (y en caso de implementar el lenguaje, mejorar la eficiencia del código ejecutable). En general adoptamos el mismo símbolo de operador predefinido que existe en Hope, aunque en algunos casos preferimos el símbolo matemático tradicional (p.ej. ≤ en vez de =<). Las funciones predefinidas del lenguaje son algunas funciones aritméticas de enteros (**+**, **-**, *****), la concatenación e inversión de listas (simbolizadas respectivamente **<>** e **invertir**) y las funciones selectoras de listas (**cabeza** y **cola**) y de tuplas (**primero**, **segundo**, ..., **n-esimo**). Las funciones selectoras producen programas poco claros, así que se evitará su uso utilizando patrones en cabeceras de las ecuaciones y en términos cualificados de definiciones locales.

Cada función tiene un único parámetro y produce un único resultado. Ejemplos de aplicaciones de las funciones predefinidas son:

```
+ (5,8)  que vale  13 : entero.
invertir [1,2,3]  o  invertir([1,2,3])  que vale
                                        [3,2,1] : lista.
```

Obsérvese que, de forma contraria a nuestra intuición, la función **+** toma un solo parámetro de tipo **entero#entero**, en vez

de dos de tipo entero. El modo de proceder es equivalente: en nuestro caso formamos el par a partir de los enteros mediante una función constructora (clase de funciones que estudiaremos en el subapartado de tipos de datos) y posteriormente se aplica + al par. Del mismo modo, la función **invertir** toma un único parámetro de tipo lista. Algunas funciones se escriben, por respeto a la tradición y por comodidad, con formato infijo (p.ej. **5+8** en vez de **+(5,8)**).

Un predicado tiene el aspecto de una función cuyo rango es el tipo **booleano**. Sin embargo, desde el punto de vista de la lógica utilizada los predicados no son funciones y no podrán ser derivados con el sistema deductivo.

2.2. LENGUAJE DE ESPECIFICACION

Podemos distinguir entre especificación y programa de una función: la especificación describe el comportamiento de la función, mientras que el programa describe un método de cálculo, un algoritmo. En otras palabras, la especificación describe "qué" hace la función y el programa describe "cómo" lo hace. Como ya se mencionó en el apartado 1.1, la distinción entre especificación y programa no siempre es clara, ya que es corriente que una especificación contenga elementos computacionales. Incluso hay veces en que la especificación consta solamente de elementos computacionales, por lo que en sí misma es un programa. Debe añadirse que en algunas funciones sencillas resulta más claro el programa que la especificación.

Nuestro interés es obtener, por deducción, un programa funcional a partir de una especificación. Siguiendo la terminología del apartado 1.1, si la especificación contiene elementos no computacionales (es decir, puramente declarativos), hablamos de síntesis de un programa a partir de la especificación. Si la especificación es puramente computacional, habla-

mos de optimización del programa. En general englobamos ambos procesos bajo el término derivación de programas.

Dado que la derivación de un programa a partir de una especificación se va a hacer de forma deductiva, es natural que el lenguaje de especificación sea un subconjunto de la lógica multigénero de predicados de primer orden. (A partir de ahora, hablaremos de lógica de predicados o simplemente de lógica para referirnos a la lógica multigénero de predicados de primer orden.) Para que la exposición sea breve, los conceptos lógicos utilizados se resumen en el Apéndice A.

SENTENCIA DE ESPECIFICACION

La especificación el comportamiento de una función (parcial) implica la existencia de dos informaciones [Bibel80, MaWa80, Smith83], como mencionamos en el apartado 1.2. En primer lugar, la función sólo está definida para un subconjunto de los valores de entrada posibles, identificables por satisfacer cierta condición, llamada condición de entrada. En segundo lugar, la función transforma valores de entrada en valores de salida, estableciendo una relación entre ellos, llamada condición de entrada-salida o, simplemente, de salida.

El lenguaje de especificación es el subconjunto de la lógica que permite expresar el comportamiento de una función mediante la información antes descrita. En concreto, el lenguaje de especificación es el conjunto de sentencias de lógica de predicados que tienen el siguiente esquema:

$$(\forall\, x{:}D)\ (\exists\, \bar{z}{:}\bar{R})\ Q[x;\bar{z}]$$

o más explícitamente:

$$(\forall\, x{:}D)\ (\exists\, \bar{z}{:}\bar{R})\ \left[\begin{array}{l}\text{if } E[x] \\ \text{then } S[x;\bar{z}]\end{array}\right]$$

donde **E[x]** y **S[x;$\bar{z}$]** son dos sentencias cualesquiera que respectivamente expresan la condición de entrada y de salida de una función **f**. Obsérvese que si la función es total, la condición de entrada es la condición obvia **true**, resultando **Q[x;$\bar{z}$]** igual a **S[x;$\bar{z}$]**.

La sentencia **Q[x;$\bar{z}$]** es una especificación de un conjunto de funciones $f_1, f_2, \ldots, f_n$. Asimismo $\bar{z}$ representa el conjunto de variables libres $z_1, z_2, \ldots, z_n$, donde **x** y todas las z_i son variables distintas entre sí. La variable **x** representa el parámetro formal de las funciones f_j, común a todas ellas. Cada variable z_j representa el resultado de la función respectiva f_j. También llamamos variable de entrada a **x** y variables de salida a las variables pertenecientes a $\bar{z}$. (Para realzar su distinto papel, separamos informalmente la variable de entrada y las variables de salida en las sentencias mediante una signo de punto y coma. También pueden separarse informalmente con punto y coma variables de salida de tipos distintos. Si existe una única variable de salida se suprime el subíndice, dejando **z** en lugar de z_1.)

Una especificación de función describe su comportamiento con el vocabulario de cierta teoría lógica. Es decir, la especificación se construye con los símbolos de predicado, constante y función pertenecientes al vocabulario de la teoría. En general suponemos que utilizamos la teoría combinada de enteros, tuplas y listas de enteros, dado que son los tipos predefinidos del lenguaje. En algunos casos se añadirá alguna teoría adicional, p.ej. conjuntos de enteros. Los símbolos de función f_j de una especificación deben ser nuevos, es decir, distintos de los símbolos pertenecientes al vocabulario de la teoría.

Ejemplo (especificación frente-ultimo).

Sea un programa **frente-ultimo** formado por dos funciones, **frente** y **ultimo**, que, dada una lista no vacía, devuelven res-

pectivamente la lista sin su último elemento y el último elemento de la lista. Su especificación es:

$$(\forall\ l{:}lista)\ (\exists\ z_1{:}lista; z_2{:}entero)\ \left[\begin{array}{l}\text{if not } l = nil\\ \text{then } l = z_1 <> z_2 ::nil\end{array}\right]$$

La especificación indica que existe una variable de entrada **l** y dos variables de salida z_1 y z_2, correspondientes a los valores devueltos por las funciones **frente** y **ultimo**, respectivamente. La condición de entrada indica que la variable de entrada no es la lista vacía, mientras que la condición de entrada-salida indica que **l** debe ser igual al resultado de concatenar z_1 con la lista atómica formada por z_2.

Una especificación ligeramente distinta es:

$$(\forall\ l{:}lista)\ (\exists\ z{:}lista\#entero)\ \left[\begin{array}{l}\text{if not } l = nil\\ \text{then } l = primero\ z <> segundo\ z ::nil\end{array}\right]$$

Esta sentencia especifica el comportamiento de una única función, que llamamos **frenult**. La función devuelve, dada una lista no vacía, el par formado por el frente y el último elemento de la lista. ∎

ESPECIFICACION DE FUNCION CON DOMINIO TUPLA

Algunas sentencias de especificación se cuantifican sobre un parámetro de tipo tupla. En esta situación es frecuente que no nos interese el dato global sino sus componentes, que pueden accederse mediante funciones selectoras. El uso continuo de éstas en una sentencia de especificación produce una sentencia oscura.

Ejemplo (especificación div-mod).

Sean dos funciones **div** y **mod** que, dado un par de enteros, devuelven respectivamente su cociente y su resto. Una especi-

ficación conjunta de las dos funciones es:

$$(\forall\ x{:}\text{entero}\#\text{entero})\ (\exists\ z_1,z_2{:}\text{entero})\ \left[\begin{array}{l}\text{if segundo } x > 0\\ \text{then primero } x = z_1*\text{segundo } x+z_2\\ \qquad \text{and } z_2 < \text{segundo } x\end{array}\right]$$

Obsérvese que el uso de las funciones selectoras en pares **primero** y **segundo** hace la sentencia un poco farragosa. ∎

Vamos a realizar una formulación equivalente, pero más clara, de la sentencia de especificación de funciones con dominio tupla. En el apartado 3.7 se hace una simplificación adicional de la especificación de funciones con rango tupla.

Proposición (sentencia de especificación simplificada).

Una sentencia de especificación con el formato:

$$(\forall\ x{:}D_1\#\ldots\#D_m)\ (\exists\ \bar{z}{:}\bar{R})\ \left[\begin{array}{l}\text{if } E[x]\\ \text{then } S[x;\bar{z}]\end{array}\right]$$

donde $D_1\#\ldots\#D_m$ es un tipo m-tupla y **R** es un tipo cualquiera, es equivalente a la sentencia con formato:

$$(\forall\ x_1{:}D_1;\ldots;x_m{:}D_m)\ (\exists\ \bar{z}{:}\bar{R})\ \left[\begin{array}{l}\text{if } E[(x_1,\ldots,x_m)]\\ \text{then } S[(x_1,\ldots,x_m);\bar{z}]\end{array}\right]$$

Demostración:

La sentencia de especificación original

$$(\forall\ x{:}D_1\#\ldots\#D_m)\ (\exists\ \bar{z}{:}\bar{R})\ \left[\begin{array}{l}\text{if } E[x]\\ \text{then } S[x;\bar{z}]\end{array}\right]$$

es equivalente (por distribución de cuantificadores) a:

$$(\forall\ x{:}D_1\#\ldots\#D_m)\ \left[\begin{array}{l}\text{if } E[x]\\ \text{then } (\exists\ \bar{z}{:}\bar{R})\ S[x;\bar{z}]\end{array}\right]$$

equivalente (por el axioma de descomposición de la teoría

de m-tuplas y lógica proposicional) a:

$$(\forall\ x{:}D_1\#\ldots\#D_m)\ \left[\begin{array}{l}\text{if } (\exists\ x_1{:}D_1;\ldots;x_m{:}D_m)\ x=(x_1,\ldots,x_m)\\ \text{then if } E[x]\\ \qquad\text{then } (\exists\ \bar{z}{:}\bar{R})\ S[x;\bar{z}]\end{array}\right]$$

equivalente (por distribución de cuantificadores) a:

$$(\forall\ x{:}D_1\#\ldots\#D_m;x_1{:}D_1;\ldots;x_m{:}D_m)\left[\begin{array}{l}\text{if } x=(x_1,\ldots,x_m)\\ \text{then if } E[x]\\ \qquad\text{then } (\exists\ \bar{z}{:}\bar{R})\ S[x;\bar{z}]\end{array}\right]$$

equivalente (por la proposición de reemplazamiento universal) a:

$$(\forall\ x_1{:}D_1;\ldots;x_m{:}D_m)\ \left[\begin{array}{l}\text{if } E[(x_1,\ldots,x_m)]\\ \text{then } (\exists\ \bar{z}{:}\bar{R})\ S[(x_1,\ldots,x_m);\bar{z}]\end{array}\right]$$

equivalente (por distribución de cuantificadores) a:

$$\begin{array}{l}(\forall\ x_1{:}D_1;\ldots;x_m{:}D_m)\\ (\exists\ \bar{z}{:}\bar{R})\end{array}\ \left[\begin{array}{l}\text{if } E[(x_1,\ldots,x_m)]\\ \text{then } S[(x_1,\ldots,x_m);\bar{z}]\end{array}\right]$$

■

La nueva sentencia de especificación resulta más expresiva porque hace explícita la estructura de la variable de entrada. Además, el reemplazamiento de **x** por $(x_1,\ldots,x_m)$ permite simplificar (por los axiomas de las funciones selectoras) cada subtérmino **primero x**, ..., **m-esimo x** por x_1, ..., x_m, respectivamente. Supondremos que estas simplificaciones se hacen automáticamente al formular la especificación nueva. Por analogía con la sentencia de especificación original, llamaremos variables de entrada o parámetros a las variables $x_1,\ldots,x_m$, aunque realmente las funciones especificadas tienen un solo parámetro **x** de tipo m-tupla.

Por sencillez las sentencias de formato:

$$\begin{array}{l}(\forall\ x_1{:}D_1;\ldots,x_m{:}D_m)\\ (\exists\ \bar{z}{:}\bar{R})\end{array}\ \left[\begin{array}{l}\text{if } E[(x_1,\ldots,x_m)]\\ \text{then } S[(x_1,\ldots,x_m);\bar{z}]\end{array}\right]$$

las escribiremos con frecuencia como:

$$(\forall\ \bar{x}:\bar{D})\ \left[\ \text{if } E[\bar{x}]\ \right]$$
$$(\exists\ \bar{z}:\bar{R})\ \left[\ \text{then } S[\bar{x};\bar{z}]\ \right]$$

Ejemplo (especificación simplificada div-mod).

Aplicando la proposición previa y simplificando todo lo posible, las funciones **div** y **mod** pueden especificarse mediante la sentencia:

$$(\forall\ x_1,x_2:\text{entero})\ \left[\ \text{if } x_2>0\ \right]$$
$$(\exists\ z_1,z_2:\text{entero})\ \left[\ \text{then } x_1 = z_1*x_2+z_2 \text{ and } z_2<x_2\ \right]$$

Pueden formularse proposiciones análogas a la anterior para parámetros de otros tipos estructurados, pero *no* lo hacemos porque su uso es mucho más infrecuente.

ESPECIFICACION COMPLETA Y PARCIAL

En algunos casos nos interesa especificar el comportamiento de una función **f** de tipo $D_1\#\ldots\#D_m\text{->}R$ (para $m\geq 1$) cuando ciertos parámetros se construyen de una manera fija, de entre las posibles según su tipo. Una especificación de este estilo especifica, no la función completa, sino una de las ecuaciones (con el patrón asociado a la forma de construir los parámetros) que van a formar el programa de la función.

Definición (especificación parcial).

Sea una especificación con el formato usual:

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\ Q[\bar{x};\bar{z}]$$

o, más explícitamente:

$$(\forall\ x_1:D_1;\ldots;x_i:D_i;\ldots;x_m:D_m)\ (\exists\ \bar{z}:\bar{R})$$
$$Q[(x_1,\ldots,x_i,\ldots,x_m);\bar{z}]$$

Denominamos especificación parcial de la especificación anterior a una sentencia con la forma:

$$(\forall\ \bar{x}':\bar{D}')\ (\exists\ \bar{z}:\bar{R})\ Q[s[\bar{x}'];\bar{z}]$$

o, más explícitamente:

$$(\forall\ x_1:D_1;\ldots;\bar{y}:\bar{T};\ldots;x_m:D_m)\ (\exists\ \bar{z}:\bar{R})\ Q[(x_1,\ldots,t[\bar{y}],\ldots,x_m);\bar{z}]$$

en algunos parámetros x_i (para $1 \leq i \leq m$), donde t es término de tipo D_i formado exclusivamente por constructores y las variables $\bar{y}$.

Una especificación que no es parcial es una especificación completa. ∎

Ejemplo (especificación parcial frenult)

Recordemos que una especificación completa de la función **frenult** es:

```
(∀ l:lista)          [ if not l=nil
(∃ z:lista#entero)   [ then l = primero z<>segundo z::nil ]
```

Incluimos tres especificaciones parciales, correspondientes a los casos en que la lista parámetro contiene respectivamente cero, uno o más enteros:

```
(∃ z:lista#entero)   [ if not nil=nil
                     [ then nil = primero z<>segundo z::nil ]

(∀ x:entero)         [if not x::nil=nil
(∃ z:lista#entero)   [then x::nil=primero z<>segundo z::nil ]

(∀ x,y:entero;l:lista)
(∃ z:lista#entero)  [if not x::y::l=nil
                    [then x::y::l=primero z<>segundo z::nil ]
```

∎

2.3. DERIVACION DEDUCTIVA DE PROGRAMAS

El proceso de derivación deductiva, muy poco desarrollado en el apartado 1.2, describe el paso de una especificación a un programa, así como la relación que hay entre ambos.

Para la derivación de un programa dependemos de que la especificación sea correcta, es decir, que describa exactamente el comportamiento deseado. Su desarrollo no es una tarea fácil en general; de hecho el desarrollo de especificaciones constituye un campo importante de la ingeniería de software llamado ingeniería de requisitos. Es corriente que durante la derivación del programa se encuentre algún fallo en la especificación inicial, debiendo modificarse y rehacer la derivación. Sin embargo, en lo sucesivo suponemos que la especificación se ha desarrollado correctamente.

PROCESO DERIVADOR

Un programa se obtiene como un producto de la demostración de una sentencia de especificación, con el formato usual:

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\ Q[\bar{x};\bar{z}]$$

La sentencia de especificación se demuestra en la teoría correspondiente de una manera constructiva, es decir, indicando un método para encontrar los valores de salida de las funciones. En consecuencia el programa derivado tiene el mismo esquema que la demostración de la especificación. Diferentes programas correspondientes a una misma especificación provienen de demostraciones alternativas. Asimismo un programa puede satisfacer varias especificaciones.

Un programa derivado satisface su especificación por especificación, es decir, se satisface el teorema de corrección:

$$\text{if } (\forall\, \bar{x}{:}\bar{D})\ P[\bar{x}]$$
$$\text{then } (\forall\, \bar{x}{:}\bar{D})\ Q[\bar{x};\bar{f}\ \bar{x}]$$

donde el antecedente representa el conjunto de igualdades cuantificadas que definen en la teoría las funciones $\bar{f}$ (en el apartado 2.4 describimos cómo convertir ecuaciones funcionales en sentencias). El teorema anterior puede añadirse al conjunto de teoremas de la teoría. De manera equivalente, si ampliamos la teoría añadiendo cada una de las igualdades de $P[\bar{x}]$ como un nuevo axioma, entonces añadimos como teorema el consecuente $(\forall\ \bar{x}{:}\bar{D})\ \ Q[\bar{x},\bar{f}\ \bar{x}]$. La incorporación de las igualdades de $P[\bar{x}]$ aumenta el conjunto de símbolos de función de la teoría con los símbolos $\bar{f}$.

Ejemplo (programa div-mod).

Supongamos que al demostrar la especificación **div-mod** del apartado 2.2 se deriva el siguiente programa miniHope:

```
dec div, mod : entero#entero -> entero ;
--- div (x1,x2)
       <= if x1<x2 then 0 else div(x1-x2,x2)+1 ;
--- mod (x1,x2)
       <= if x1<x2 then x1 else mod(x1-x2,x2) ;
```

Podemos ampliar la teoría de enteros con dos nuevos axiomas que expresan el programa anterior:

$$(\forall\ x_1,x_2{:}\text{entero})\ \text{div}(x_1,x_2) = \left[\begin{array}{l}\text{if } x_1<x_2\\ \text{then } 0\\ \text{else } \text{div}(x_1-x_2,x_2)+1\end{array}\right]$$

$$(\forall\ x_1,x_2{:}\text{entero})\ \text{mod}(x_1,x_2) = \left[\begin{array}{l}\text{if } x_1<x_2\\ \text{then } x_1\\ \text{else } \text{mod}(x_1-x_2,x_2)\end{array}\right]$$

En la nueva teoría ampliada se cumple la sentencia de *corrección de **div** y **mod**:*

$$(\forall\ x_1,x_2{:}\text{entero})\ \left[\begin{array}{l}\text{if } x_2>0\\ \text{then } x_1=\text{div}(x_1,x_2)*x_2+\text{mod}(x_1,x_2)\\ \qquad \text{and } \text{mod}(x_1,x_2)<x_2\end{array}\right]$$

que por tanto puede añadirse como un teorema. ∎

Ejemplo (programa frenult).

Sea el siguiente programa miniHope derivado como consecuencia de la demostración de la especificación **frenult** incluida en el apartado 2.2:

```
dec frenult : lista -> lista#entero ;
--- frenult x::nil
      <= (nil,x) ;
--- frenult x::y::l
      <= let (f,u) == frenult (y::l)
         in (x::f,u) ;
```

Ampliamos la teoría combinada de enteros, listas y pares de ambos con los dos axiomas correspondiente a las ecuaciones anteriores:

```
(∀ x:entero)  frenult x::nil = (nil,x)

(∀ x,y:entero;l:lista)
[                frenult x::y::l                          ]
[                        =                                ]
[ (x::primero frenult(y::l),segundo frenult(y::l))        ]
```

En la teoría ampliada resultante se cumple la sentencia de corrección de **frenult**:

```
(∀ l:lista)
[ if not l = nil                                          ]
[ then l = primero frenult l<>segundo frenult l::nil      ]
```

∎

Una forma de conseguir funciones con patrones es traducir un programa derivado en estilo Lisp en otro equivalente con

patrones; la traducción puede hacerse automáticamente, al menos en un gran número de casos. Sin embargo, de esta forma el esfuerzo requerido para derivar funciones con patrones es mayor que el requerido para derivar funciones sin patrones. Si se tiene en cuenta que la programación directa con patrones resulta más fácil a las personas, es razonable esperar que el proceso deductivo también deba ser más sencillo si se incorpora el uso de patrones. Esto obliga a identificar, en el resto del capítulo, los fundamentos lógicos necesarios, que en el Capítulo 3 son utilizados para modificar el sistema deductivo de Manna y Waldinger.

SIMBOLOS PRIMITIVOS

A veces interesa que algún símbolo computable no aparezca en el programa por derivar: es un caso frecuente en optimización de funciones, donde queremos obtener un programa que no utilice ciertos símbolos que introducen ineficiencia. Tampoco deben aparecer símbolos no computables, como cuantificadores o símbolos de Skolem. Esta restricción se consigue distinguiendo entre símbolos primitivos y no primitivos; sólo los símbolos primitivos pueden aparecer en el cuerpo de una función. Si no deseamos que un símbolo particular aparezca en una función, se declara no primitivo antes de la derivación; por tanto, el conjunto de funciones primitivas varía de derivación en derivación. Los símbolos de las funciones por derivar son automáticamente primitivos para permitir la derivación de funciones recursivas.

En el ejemplo anterior sobre **frenult** las funciones "coma", **::** y **frenult** se consideran primitivas. No ocurre lo mismo con las funciones selectoras de tuplas **primero** y **segundo**, lo que se resuelve mediante una declaración local con patrones.

ENDULZAMIENTO DE ESPECIFICACIONES

El uso de la lógica como lenguaje de especificación tiene como ventaja su genericidad. Sin embargo, una especificación lógica no realza sus partes principales, pudiendo resultar farragosa o poco legible. En concreto, debe resaltar las variables de entrada y salida, con sus tipos, y las condiciones de entrada y de salida. Además, la especificación no incluye los símbolos de las funciones especificadas. Conseguimos estos objetivos utilizando un formato de especificación [MaWa80] más estructurado que la sola sentencia de especificación:

$\bar{f}\ \bar{x}:\bar{D}$ <= encontrar $\bar{z}:\bar{R}$
tal que $S[\bar{x};\bar{z}]$
donde $E[\bar{x}]$

Obsérvese que implícitamente se supone que las variables $\bar{x}$ están cuantificadas universalmente y las variables $\bar{z}$, existencialmente. Si $\bar{x}$ contiene más de una variable, éstas se encuentran formando una tupla, haciendo explícita la forma del parámetro.

Debe recordarse que una especificación de este estilo es una mero convenio para representar la verdadera sentencia lógica de especificación. La conversión entre ambas es inmediata.

Ejemplo (especificación endulzada frenult).

Utilizando el formato anterior, la especificación de **frenult** se formula:

```
frenult l : lista  <= encontrar  z : lista#entero
                      tal que  l=primero z<>segundo z::nil
                      donde  not l=nil
```

■

Ejemplo (especificación endulzada div-mod).

Análogamente, especificamos las funciones **div** y **mod** de la siguiente manera:

div,mod (x_1,x_2) : entero#entero
<= encontrar z_1,z_2 : entero
tal que $x_1=z_1*x_2+z_2$ and $z_2<x_2$
donde $x_2>0$

2.4. RELACION ENTRE PROGRAMAS Y TEORIAS LOGICAS

Uno de los puntos claves en la definición de un sistema transformativo es la relación entre los lenguajes de especificación y programación. En los métodos deductivos la relación se establece entre la lógica de predicados y el lenguaje de programación. De hecho esta relación define una semántica del lenguaje de programación, sirviendo para justificar la transición de términos lógicos a términos funcionales durante el proceso de derivación. Consideramos el lenguaje de programación funcional como una forma endulzada de expresar un subconjunto de la lógica. Por tanto, determinamos la relación entre ambos lenguajes dictando reglas de paso de un programa funcional a la lógica de predicados.

Hay dos partes bien diferenciadas en los programas mini-Hope: declaraciones de tipos y declaraciones de programas. Cada una de las dos partes tiene una relación directa pero diferente con la lógica de predicados: una declaración de tipos crea una nueva teoría lógica mientras que una declaración de función amplía una teoría lógica ya existente con una nueva función. (Dado que una declaración de tipos introduce un tipo nuevo, siguiendo disponibles los tipos previos, es más correcto decir que una declaración de tipos amplía con un nuevo tipo la teoría lógica combinada de los tipos existentes.

Sin embargo, será frecuente que nos refiramos a la teoría lógica de cada tipo de datos como si fuera una teoría independiente.) Veamos ambas partes por separado.

RELACION ENTRE TIPO DE DATOS Y TEORIA LOGICA

Recuérdese que las definiciones de tipos de datos en mini-Hope tienen un formato fijo:

data T == C_1 T_1 ++ ... + + C_n T_n ;

donde **T** es el símbolo del nuevo tipo, cada C_i es un constructor del tipo y cada T_i es el tipo del dominio del constructor respectivo C_i.

Los constructores de un tipo **T**, junto con sus tipos asociados, definen el conjunto de valores incluidos en el tipo **T**. Esta estructura matemática, llamada álgebra de palabras libre (o álgebra libre), tiene un conjunto de características que permite definir su teoría lógica con un formato fijo. A continuación vemos los distintos aspectos de la definición de la teoría de un tipo álgebra libre. Para ilustrarlo utilizamos la declaración del tipo lista, que repetimos:

```
data lista == nil ++ entero::lista ;
```

Tipos de los constructores

Una declaración de tipo de datos introduce constructores nuevos. Toda aparición futura de éstos en una sentencia tiene asociada su tipo respectivo: si un constructor C_i es una constante, entonces su tipo es **T** y si es una función, su tipo es T_i -> **T**.

Por ejemplo, los constructores de listas tienen los siguientes tipos:

```
nil : lista
::  : entero#lista -> lista
```

Axiomas de unicidad

Cada valor de un álgebra libre se crea de forma única a partir de constructores. Esta propiedad se declara mediante dos conjuntos de axiomas de unicidad. El primer grupo declara que los términos formados con constructores distintos también son distintos. Para ello se incluyen tantos axiomas como sean necesarios para indicar la desigualdad de dos términos formados con constructores distintos.

Por ejemplo, la teoría de listas incluye dos únicos constructores, por lo que basta con el axioma:

```
(∀ x:entero;l:lista)  not nil=x::l
```

El segundo grupo de axiomas declara que dos términos formados por la aplicación de una misma función constructora son iguales sólo si son iguales los respectivos subtérminos a los que se aplica. La igualdad de estos subtérminos puede expresarse de la siguiente manera. Si los términos son variables, se incluye una igualdad de las dos variables. Si los términos son tuplas de **n** variables, se incluyen **n** igualdades, cada una comparando las variables respectivas de las tuplas.

Por ejemplo, la teoría de listas sólo incluye una función constructora, por lo que basta con el axioma:

```
(∀ x1,x2:entero)   [ if x1::l1=x2::l2      ]
(∀ l1,l2:lista)    [ then x1=x2 and l1=l2  ]
```

Axiomas de igualdad

Un álgebra de palabras es una teoría **T** con igualdad, es decir, donde existe un predicado de igualdad $=_T$ que satisface

los axiomas de igualdad. Cada nueva teoría **T** introduce un nuevo predicado $=_T$, pero de una manera informal representamos todos los predicados de igualdad con el mismo símbolo =.

Los axiomas de igualdad incluyen la reflexividad, simetría y transitividad:

(∀ x:T) x=x

(∀ x,y:T) if x=y then y=x

(∀ x,y,z:T) if x=y and y=z then x=z

Además deben incluirse las instancias necesarias de los esquemas de sustitutividad funcional y sustitutividad en predicados.

Por ejemplo, en la teoría de listas, los tres primeros axiomas son como siempre:

(∀ l:lista) l=l

(∀ l1,l2:lista) if l1=l2 then l2=l1

(∀ l1,l2,l3:lista) if l1=l2 and l2=l3 then l1=l3

Dado que la única función de la teoría básica de listas es el constructor ::, cuyo parámetro es un par, deben incluirse dos instancias del esquema de sustitutividad funcional, uno por cada elemento del par:

(∀ x,y:entero;l:lista) if x=y then x::l=y::l

(∀ x:entero;l1,l2:lista) if l1=l2 then x::l1=x::l2

La adición posterior de nuevas funciones o predicados a esta teoría básica de listas obliga a la inclusión de nuevas instancias de los esquemas de sustitutividad por igualdad.

Principio de inducción estructural

Si la declaración de tipos es recursiva, debe proporcionarse algún medio para razonar sobre la estructura recursiva de los datos. Para esto se incluye un principio de inducción estructural. El adjetivo "estructural" refleja que el principio permite demostrar sentencias utilizando directamente la estructura (recursiva) de los datos; esto implica considerar tantos casos como formas posibles haya de construirlos. Esta forma de inducción es conocida en programación [Burstall69, MaWa85].

El principio de inducción estructural para un tipo **T** se formula de la siguiente forma general. Sea una sentencia cualquiera cuantificada sobre una variable de tipo **T**, es decir, una sentencia con la forma **(∀ x:T) F[x]**. La variable **x** se llama la variable de inducción y la sentencia **F[x]** se llama la sentencia inductiva. Para demostrar su validez basta demostrar, para cada constructor C_i del tipo **T**:

- Si C_i es un constructor constante, entonces $F[C_i]$ es válida. Cada sentencia $F[C_i]$ se llama un caso base.
- Si C_i es un constructor con dominio de tipo $T_1\#...\#T_n$, con $n>0$, entonces se cumple $F[C_i(x_1,...,x_n)]$ supuesto que se cumple $F[x_k]$ para aquellas variables x_k de tipo **T**. Las sentencias $F[x_k]$ y $F[C_i(x_1,...,x_n)]$ se llaman respectivamente las hipótesis de inducción y la conclusión deseada. La implicación de ambas sentencias se llama un paso inductivo.

Dicho principio de inducción se formula como una implicación cuyo antecedente es la conjunción de las condiciones anteriores y cuyo consecuente es la sentencia a demostrar **(∀ x:T) F[x]**. Obsérvese que el principio de inducción no es un axioma, sino un esquema de axioma, por lo que es sólo al demostrar una sentencia **(∀ x:T) F[x]** concreta cuando se introduce como axioma la instanciación correspondiente del princi-

pio.

Por ejemplo, el principio de inducción estructural para listas es una implicación cuyo antecedente tiene dos conjuntados, uno por constructor. Su formulación es:

```
if [ F[nil]
     and
   (∀ x:entero;l:lista) [ if F[l]
                          [ then F[x::l] ] ]
then (∀ l:lista) F[l]
```

En este principio, la variable inductiva es **l**, la sentencia inductiva es **F[l]**, el caso base es el primer conjuntado del antecedente y el caso inductivo es el segundo conjuntado.

Teoremas adicionales

Podemos añadir a la teoría cualesquiera sentencias válidas en la teoría, es decir, teoremas de la teoría. Por ejemplo, un teorema que es importante en los dos próximos apartados es el teorema de descomposición. Dicho teorema afirma que un término de tipo **T** se forma con alguno de los constructores del tipo.

Por ejemplo, el teorema de descomposición de listas dice que una lista es bien la lista vacía bien una lista formada con **::** a partir de un entero y una sublista. Su formulación es:

```
(∀ l:lista) [          l=nil
            [            or
            (∃ x:entero;l':lista) l=x::l' ]
```

Otras teorías

La definición de tipo de datos a partir de un álgebra de palabras es un instrumento adecuado para un buen número de casos. Sin embargo, hay tipos de datos que no se corresponden exactamente con los definidos mediante constructores. Por ejemplo, es corriente que no todos los valores generados por

el álgebra de palabras estén permitidos o incluso que no tengan sentido. No hay reglas fijas para resolver estos casos satisfactoriamente. En programación es corriente el uso de tipos abstractos de datos, donde se esconde la implementación y se deja visible un conjunto de operaciones sobre el tipo.

Para permitir la existencia de cualquier teoría lógica adoptamos una solución modesta, consistente en seguir dos pasos. Por un lado realizamos una declaración normal de tipo de datos utilizando constructores. Posteriormente podemos optar por considerar al nuevo tipo como un álgebra libre o darle otra estructura lógica. En el primer caso, la definición de su teoría es automática siguiendo las reglas vistas hasta ahora. En el segundo caso, debemos construir la teoría siguiendo nuestro sentido común. Los únicos puntos, de los utilizados para las álgebras libres, que se conservan son la asociación automática de tipos a los constructores utilizados y la inclusión de los axiomas de la igualdad.

Por ejemplo, sea el tipo de los conjuntos de enteros. Su definición como álgebra libre no es satisfactoria. Así, el conjunto formado al añadir un elemento a un conjunto debe ser el mismo que el formado al añadir dos veces el elemento, pero en un álgebra libre los axiomas de unicidad harían que se considerasen conjuntos distintos.

En primer lugar realizamos una declaración de tipos:

data conjunto == ϕ ++ entero.conjunto ;

supuesto que pudiera utilizarse ϕ como símbolo de constante. A partir de esta definición sabemos que ϕ y . tienen de tipos respectivos **conjunto** y **entero#conjunto -> conjunto**. Para no alargar demasiado la exposición, remitimos al lector interesado en la teoría de conjuntos al apartado B.4.

RELACION ENTRE DECLARACION DE FUNCION Y AXIOMAS

Una declaración de función no introduce ninguna teoría nueva, ya que no añade ningún nuevo tipo de datos. Su efecto lógico consiste en ampliar la teoría existente con funciones nuevas.

Un programa miniHope consta de un conjunto de declaraciones de función, cada una formada por un conjunto de ecuaciones. El formato general es:

```
dec f1 : D1 -> R1 ;
--- f1 p11 <= t11 ;
   ...
--- f1 p1n1 <= t1n1 ;
   ...
dec fm : Dm -> Rm ;
--- fm pm1 <= tm1 ;
   ...
--- fm pmnm <= tmnm ;
```

donde cada f_i ($1 \leq i \leq m$) representa una función de dominio D_i y rango R_i y la ecuación j-ésima ($1 \leq j \leq n_i$) de cada función f_i tiene patrón p_{ij} y lado derecho t_{ij}.

La declaración de tipo de cada función f_i ($1 \leq i \leq m$) asocia el tipo $D_i \rightarrow R_i$ a dicho símbolo de función.

Cada ecuación del programa produce la introducción de un axioma. Cada axioma es una igualdad de dos términos, cuantificada universalmente sobre las variables libres. Los dos lados de la igualdad son el lado izquierdo y derecho de la ecuación respectiva, excepto en lo respectivo a las definiciones locales, como veremos a continuación. Por tanto, el conjunto de sentencias tiene el formato:

$(\forall\ \bar{x}_{11}:\bar{D}_{11})\ f_1\ p_{11} = t_{11}'$

$\vdots$

$(\forall\ \bar{x}_{1n1}:\bar{D}_{1n1})\ f_1\ p_{1n1} = t_{1n1}'$

$\vdots$

$(\forall\ \bar{x}_{m1}:\bar{D}_{m1})\ f_m\ p_{m1} = t_{m1}'$

$\vdots$

$(\forall\ \bar{x}_{mnm}:\bar{D}_{mnm})\ f_m\ p_{mnm} = t_{mnm}'$

El tipo $\bar{D}_{ij}$ de las variables $\bar{x}_{ij}$ de cada axioma se obtiene a partir de la declaración de tipos de la función f_i y el patrón de la ecuación respectiva utilizando las reglas de inferencia de tipos del lenguaje.

Cada término t_{ij}' es el equivalente lógico del término funcional t_{ij}. La correspondencia entre términos del lenguaje de programación y de la lógica se establece siguiendo las siguientes reglas, según la forma de t_{ij}:

- t_{ij} no contiene definiciones locales. Entonces el término está formado según la sintaxis de términos de la lógica de predicados. Como además los símbolos de función y predicado coinciden en ambos lenguajes, t_{ij}' coincide con t_{ij}.

- t_{ij} contiene definiciones locales de variables, es decir, definiciones locales sin patrones. Entonces el término t_{ij} contiene algún subtérmino **s** con la forma

  let v == s^1 in s^2

  En este caso la variable local **v** se utiliza como una abreviatura del término s^1 en s^2. Es decir, el término **s** es equivalente al término resultante de reemplazar en el término principal s^2 cada aparición de la variable local **v** por el término cualificado s^1. Si representamos t_{ij} como t_{ij}[let v==s^1 in s^2[v]], entonces t_{ij}' es

  $t_{ij}[s^2[s^1]]$.

- t_{ij} contiene definiciones locales con patrones. Veamos el caso más frecuente, cuando el patrón es una tupla. (La aparición de otro constructor se trata de forma

similar.) Entonces el término t_{ij} contiene algún subtérmino **s** con la forma

$$\text{let } (v_1,\ldots,v_p) == t_{ij}{}^1 \text{ in } t_{ij}{}^2$$

En esta situación puede hallarse el valor de cada variable con ayuda de las funciones selectoras sobre tuplas; p.ej. v_1 tiene como valor **primero** $t_{ij}{}^1$. Es decir, el término **s** es equivalente al término resultante de reemplazar simultáneamente en el término principal s^2 cada aparición de cada variable local v_k ($1 \leq k \leq p$) por **k-ésimo** s^1. Si representamos t_{ij} como t_{ij}**[let** $(v_1,\ldots,v_p)==s^1$ **in** $s^2[v_1,\ldots,v_p]]$, entonces t_{ij}' es

$$t_{ij}[s^2[\textbf{primero } s^1,\ldots,\textbf{p-ésimo } s^1]].$$

Suponemos que el término t'_{ij} se simplifica automáticamente lo máximo posible. Por ejemplo, cualquier término **primero** $(t_1,\ldots)$ se simplifica a t_1.

Si t_{ij} contiene varias definiciones locales, el proceso de reescritura se realiza repetidamente en cualquier orden (p.ej. de afuera hacia adentro o viceversa siguiendo la estructura sintáctica del término t_{ij}) hasta que obtenemos un término sin definiciones locales.

La conversión anterior se justifica a partir de las reglas de evaluación (ver apartado A.7) y los axiomas de las funciones selectoras sobre tuplas (ver apartado B.2).

En el apartado 2.3 se incluyen dos ejemplos de programa, **div-mod** y **frenult**, con sus dos axiomas asociados.

EVALUACION DE TERMINOS

Dado un programa funcional, podemos evaluar cualquier término básico, es decir, hallar su valor. La evaluación de un término es un proceso repetitivo de reescritura de subtérminos por otros iguales. El proceso es conocido en programación funcional, por lo que no lo repetimos aquí. Nuestro lenguaje uti-

liza un conjunto de reglas de evaluación que definen una estrategia ambiciosa. Su semántica correspondiente es una semántica estricta.

La evaluación de un término tiene una clara correspondencia lógica. Sea un término cualquiera **t:T** a evaluar. Su evaluación corresponde a la demostración del teorema **(∃ z:T) z=t.** La demostración se realiza mediante repetidas sustituciones por igualdad de subtérminos de **t** por otros iguales, según los axiomas (las ecuaciones) de sus funciones, y una aplicación final del axioma reflexivo de la igualdad, obteniendo un valor básico para **z**. Una generalización de este mecanismo a términosno básicos permite optimizar funciones ya existentes, como se hace en el método de despliegue-pliegue [BuDa77] y en la derivación del apartado 4.2.

RELACION ENTRE RECURSION E INDUCCION

En programación funcional la repetición de cierta operaciónse hace mediante recursión. La derivación por deducción de una función recursiva precisa algún mecanismo repetitivo similar en la teoría lógica: un principio de inducción.

Cada esquema de recursión obliga al uso de cierto principio de inducción durante el proceso derivador. Veamos como ejemplo los esquemas de programa obtenidos gracias al principio de inducción estructural, que llamaremos programas recursivos estructurles. Sea una función **f** con el parámetro de tipo **T** o **...#T#...**, siendo la declaración del tipo **T** como de costumbre:

data T == C_1 T_1 ++ ... ++ C_n T_n ;

donde algún T_i es una tupla donde aparece **T**, es decir, **T** es un tipo recursivo. Utilizando el principio de inducción estructural sobre **T** podemos derivar, para cierta especificación de **f**

un programa de **n** ecuaciones, una por constructor. Si el constructor C_i se aplica a parámetros de tipo **T**, la ecuación tiene la forma:

$$\text{--- f } C_i \text{ (...,}\bar{x}_{ij}\text{,...) <= } t_i[f\ \bar{x}_{ij}] \text{ ;}$$

donde las variables $\bar{x}_{ij}$ tienen tipo **T**. O si el constructor C_i no se aplica a parámetros de tipo **T**, tiene la forma:

$$\text{--- f } C_i\ t_i \text{ <= } s_i \text{ ;}$$

donde **f** no aparece en s_i.

Por ejemplo, en una teoría de listas, dicho esquema de función produciría la siguiente declaración:

$$\text{--- f (...,nil,...) <= } t_1 \text{ ;}$$
$$\text{--- f (...,x::l,...) <= } t_2 \text{ [f l] ;}$$

No todas las funciones con algún parámetro de tipo lista se ajustan de manera natural a este formato. Por tanto se precisan otros esquemas de inducción, aunque no hay que asustarse, porque hay pocos principios de inducción prácticos [Bibel80, MaWa80]. De hecho basta con el principio de inducción bien fundada, pero la definición de programas por patrones concede una importancia especial a la inducción estructural. Un teorema no inductivo que también es importante en la derivación de funciones con patrones es la propiedad de descomposición. Ambos principios lógicos son objeto de tratamiento en los dos siguientes apartados.

2.5. DEMOSTRACION POR INDUCCION

En este apartado vemos diversos principios de inducción que permiten la derivación de diferentes programas recursivos: estructurales con un número cualquiera de ecuaciones, estruc-

turales sobre varios parámetros o recursivos que no siguen la estructura de los datos. Para estos casos necesitamos respectivamente versiones del principio de inducción estructural con varios casos básicos, principios de inducción estructural simultánea sobre varios componentes de una tupla y un principio de inducción bien fundada.

PRINCIPIOS DE INDUCCION ESTRUCTURAL SOBRE ENTEROS NO NEGATIVOS

El principio de inducción estructural sobre enteros (ver apartado B.1) se formula:

```
                ┌ F[0]                                    ┐
if              │   and                                   │
                │ (∀ x:entero) ┌ if F[x]                  │
                └              └ then F[succ x] ┘         ┘
then (∀ x:entero) F[x]
```

Existe otra versión del principio de inducción estructural, llamado principio de inducción estructural por descomposición, por contraposición al anterior, que razona por generación. Lo expresamos:

```
                          ┌ if x=0                        ┐
if (∀ x:entero)           │ then F[0]                     │
                          └ else ┌ if F[x-1] ┐            ┘
                                 └ then F[x] ┘
then (∀ x:entero) F[x]
```

o, en su formulación más corriente,

```
                ┌ F[0]                                          ┐
                │    and                                        │
if              │                  ┌ if not x=0                 │
                │ (∀ x:entero)     │ then ┌ if F[x-1] ┐        │
                └                  └      └ then F[x] ┘ ┘       ┘
then (∀ x:entero) F[x]
```

Ambos principios derivan programas con recursión estructural, pero mientras la distinción entre casos se realiza por

patrones utilizando el primer principio, con el segundo se efectúa mediante una expresión condicional.

Con frecuencia resulta conveniente utilizar para la derivación de programas recursivos estructurales otras versiones del principio de inducción estructural por generación que distingan varios casos básicos, no sólo el 0. Así podemos utilizar el siguiente principio de inducción estructural con dos casos básicos. (Utilizamos la terminología **x+n** para representar el término más correcto, expresado con constructores, **succ ... succ x** (**n** veces).)

```
    ⌈ F[0]                                          ⌉
    |    and                                       |
if  | F[1]                                          |
    |    and                                       |
    | (∀ x:entero) ⌈ if F[x+1]     ⌉               |
    ⌊              ⌊ then F[x+2]   ⌋               ⌋
then (∀ x:entero) F[x]
```

Generalizando, el principio de inducción estructural con **k+1** casos básicos es:

```
    ⌈ F[0]                                          ⌉
    |    and                                       |
    |     ...                                       |
if  |    and                                       |
    | F[k]                                          |
    |    and                                       |
    | (∀ x:entero) ⌈ if F[x+k]       ⌉             |
    ⌊              ⌊ then F[x+k+1]   ⌋             ⌋
then (∀ x:entero) F[x]
```

En todos los princ..pios de inducción anteriores se distinguen las partes descritas en el apartado 2.4.

Proposición (validez de principios de inducción estructural sobre enteros con distintos casos básicos).

En la teoría de enteros las distintas versiones del principio de inducción estructural con diferente número de casos básicos son válidas.

Demostración:

La demostración la hacemos por inducción sobre el número de casos básicos. La situación básica es el principio con un caso, válido porque es un axioma de la teoría de enteros.

Veamos el caso inductivo. Suponemos que el principio de inducción con **k** casos es válido:

```
    ⌈ F[0]                                          ⌉
    |    and                                        |
    |     ...                                       |
if  |    and                                        |
    | F[k-1]                                        |
    |    and                                        |
    | (∀ x:entero) ⌈ if F[x+k-1] ⌉                  |
    ⌊              ⌊ then F[x+k] ⌋                  ⌋
then (∀ x:entero) F[x]
```

Se trata de demostrar que también es válido el principio de inducción con **k+1** casos:

```
    ⌈ F[0]                                          ⌉
    |    and                                        |
    |     ...                                       |
if  |    and                                        |
    | F[k]                                          |
    |    and                                        |
    | (∀ x:entero) ⌈ if F[x+k]     ⌉                |
    ⌊              ⌊ then F[x+k+1] ⌋                ⌋
then (∀ x:entero) F[x]
```

Dado que el primer principio es válido, basta demostrar la equivalencia de ambos principios para establecer la validez del segundo. El consecuente de ambos principios es el mismo, por lo que basta demostrar la equivalencia de sus antecedentes.

El antecedente del primer principio es:

```
F[0]
  and
   ...
  and
F[k-1]
  and
(∀ x:entero) [ if F[x+k-1]
               [ then F[x+k] ]
```

equivalente (según el teorema de descomposición y lógica proposicional) a:

```
F[0]
  and
   ...
  and
F[k-1]
  and
                [ [       x=0          ] ]
(∀ x:entero)    [ [ if [      or      ] ] ]
                [ [    (∃ y:entero) x=y+1 ] ]
                [ [ then [ if F[x+k-1] ]  ] ]
                [ [      [ then F[x+k] ]  ] ]
```

equivalente (según distributividad de la implicación y los cuantificadores) a:

```
F[0]
  and
   ...
  and
F[k-1]
  and
(∀ x:entero)  [ if x=0                          ]
              [ then [ if F[x+k-1] ]            ]
              [      [ then F[x+k] ]            ]
  and
(∀ x,y:entero) [ if x=y+1                       ]
               [ then [ if F[x+k-1] ]           ]
               [      [ then F[x+k] ]           ]
```

equivalente (según la proposición de reemplazamiento universal y propiedades de los enteros) a:

```
F[0]
  and
   ...
  and
F[k-1]
  and
if F[k-1] then F[k]
  and
(∀ y:entero) [ if F[y+k]      ]
             [ then F[y+k+1] ]
```

equivalente (por lógica proposicional y renombrado de variables ligadas) a:

```
F[0]
  and
   ...
  and
F[k-1]
  and
F[k]
  and
(∀ x:entero) [ if F[x+k]      ]
             [ then F[x+k+1] ]
```

que es el antecedente del principio de inducción con **k+1** casos básicos, como queríamos obtener. ∎

PRINCIPIOS DE INDUCCION ESTRUCTURAL SOBRE LISTAS

De manera análoga a los enteros no negativos la teoría de listas (ver apartado B.3) dispone de un principio de inducción estructural por generación que afirma la validez del cierre universal de:

```
    [ F[nil]                                    ]
if  [   and                                     ]
    [ (∀ x:entero)  [ if F[l]       ]           ]
    [ (∀ l:lista)   [ then F[x::l] ]           ]
then (∀ l:lista) F[l]
```

donde **x** no aparece libre en **F[l]**. Por brevedad, en lo sucesivo no citaremos estas restricciones, que supondremos que se cumplen. También existe un principio de inducción estructural por

descomposición:

```
                  ┌ if l=nil                          ┐
if (∀ l:lista)    │ then F[nil]                       │
                  │ else ┌ if F[cola l] ┐             │
                  └      └ then F[l]    ┘             ┘
then (∀ l:lista) F[l]
```

más comúnmente expresado como:

```
    ┌ F[nil]                                                    ┐
    │   and                                                     │
if  │                    ┌ if not l=nil                       ┐ │
    │ (∀ l:lista)        │ then ┌ if F[cola l] ┐              │ │
    └                    └      └ then F[l]    ┘              ┘ ┘
then (∀ l:lista) F[l]
```

De nuevo cada principio produce un programa con recursión estructural diferente. Asimismo se necesitan diversas versiones del principio de inducción estructural por generación, cada uno con un número distinto de casos básicos. El principio con dos casos básicos es el siguiente.

```
    ┌ F[nil]                                                  ┐
    │   and                                                   │
if  │ (∀ x:entero) F[x::nil]                                  │
    │   and                                                   │
    │ (∀ x,y:entero)  ┌ if F[x::l]          ┐                │
    └ (∀ l:lista)     └ then F[y::x::l]     ┘                ┘
then (∀ l:lista) F[l]
```

Generalizando obtenemos el siguiente principio con **k+1** casos básicos:

```
    ┌ F[nil]                                                                  ┐
    │   and                                                                   │
    │    ...                                                                  │
    │   and                                                                   │
if  │ (∀ x1,...,xk:entero) F[xk::...::x1::nil]                                │
    │   and                                                                   │
    │ (∀ x1,...,xk,xk+1:entero;l:lista)                                       │
    └    ┌ if F[xk::...::x1::l]              ┐                              ┘
         └ then F[xk+1::xk::...::x1::l]      ┘
then (∀ l:lista) F[l]
```

En todos los principios de inducción anteriores se distin-

guen las partes descritas en el apartado 2.4.

Proposición (validez de principios de inducción estructural sobre listas con distintos casos básicos).

En la teoría de listas las distintas versiones del principio de inducción estructural con diferentes números de casos básicos son válidas.

Demostración:

La demostración la hacemos por inducción sobre el número de casos básicos. La situación básica es el principio con un caso, válido puesto que es uno de los axiomas de la teoría de listas.

Veamos el caso inductivo. Suponemos que el principio de inducción con **k** casos es válido:

$$
\begin{array}{l}
\left[\begin{array}{l}
\text{F[nil]} \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1}\text{:entero})\ \text{F}[x_{k-1}\text{::}\ldots\text{::}x_1\text{::nil}] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k\text{:entero;l:lista}) \\
\quad \left[\begin{array}{l}\text{if F}[x_{k-1}\text{::}\ldots\text{::}x_1\text{::l}] \\ \text{then F}[x_k\text{::}x_{k-1}\text{::}\ldots\text{::}x_1\text{::l}]\end{array}\right]
\end{array}\right] \\
\text{then } (\forall\ \text{l:lista})\ \text{F[l]}
\end{array}
$$

Se trata de demostrar que también es válido el principio de inducción con **k+1** casos.

$$
\begin{array}{l}
\left[\begin{array}{l}
\text{F[nil]} \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_k\text{:entero})\ \text{F}[x_k\text{::}\ldots\text{::}x_1\text{::nil}] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_k,x_{k+1}\text{:entero;l:lista}) \\
\quad \left[\begin{array}{l}\text{if F}[x_k\text{::}\ldots\text{::}x_1\text{::l}] \\ \text{then F}[x_{k+1}\text{::}x_k\text{::}\ldots\text{::}x_1\text{::l}]\end{array}\right]
\end{array}\right] \\
\text{then } (\forall\ \text{l:lista})\ \text{F[l]}
\end{array}
$$

Dado que el primer principio es válido, basta demostrar la equivalencia de ambos principios para establecer la validez del segundo. El consecuente de ambos principios es el mismo, por lo que basta demostrar la equivalencia de sus antecedentes.

El antecedente del primer principio es:

$$
\begin{array}{l}
F[nil] \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1}\text{:entero})\ F[x_{k-1}::\ldots::x_1::nil] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k\text{:entero})\ \left[\begin{array}{l}\text{if } F[x_{k-1}::\ldots::x_1::l] \\ \text{then } F[x_k::x_{k-1}::\ldots::x_1::l]\end{array}\right] \\
(\forall\ l\text{:lista})
\end{array}
$$

equivalente (según el teorema de descomposición y lógica proposicional) a:

$$
\begin{array}{l}
F[nil] \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1}\text{:entero})\ F[x_{k-1}::\ldots::x_1::nil] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k\text{:entero;l:lista}) \\
\quad \left[\begin{array}{l}\text{if } \left[\begin{array}{l} l=nil \\ \text{or} \\ (\exists\ x\text{:entero;l':lista})\ l=x::l'\end{array}\right] \\ \text{then } \left[\begin{array}{l}\text{if } F[x_{k-1}::\ldots::x_1::l] \\ \text{then } F[x_k::x_{k-1}::\ldots::x_1::l]\end{array}\right]\end{array}\right]
\end{array}
$$

equivalente (según distributividad de la implicación y los cuantificadores) a:

$$
\begin{array}{l}
F[nil] \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1}\text{:entero})\ F[x_{k-1}::\ldots::x_1::nil] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k\text{:entero;l:lista}) \\
\quad \left[\begin{array}{l}\text{if } l=nil \\ \text{then } \left[\begin{array}{l}\text{if } F[x_{k-1}::\ldots::x_1::l] \\ \text{then } F[x_k::x_{k-1}::\ldots::x_1::l]\end{array}\right]\end{array}\right]
\end{array}
$$

$$
\begin{array}{l}
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k,x:\text{entero};l,l':\text{lista}) \\
\quad \left[\begin{array}{l} \text{if } l=x::l' \\ \text{then } \left[\begin{array}{l} \text{if } F[x_{k-1}::\ldots::x_1::l] \\ \text{then } F[x_k::x_{k-1}::\ldots::x_1::l] \end{array}\right] \end{array}\right]
\end{array}
$$

equivalente (según la proposición de reemplazamiento universal) a:

$$
\begin{array}{l}
F[nil] \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1}:\text{entero})\ F[x_{k-1}::\ldots::x_1::nil] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k:\text{entero})\ \left[\begin{array}{l} \text{if } F[x_{k-1}::\ldots::x_1::nil] \\ \text{then } F[x_k::x_{k-1}::\ldots::x_1::nil] \end{array}\right] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k,x:\text{entero};l':\text{lista}) \\
\quad \left[\begin{array}{l} \text{if } F[x_{k-1}::\ldots::x_1::x::l'] \\ \text{then } F[x_k::x_{k-1}::\ldots::x_1::x::l'] \end{array}\right]
\end{array}
$$

equivalente (por lógica proposicional y de predicados y renombrado de variables ligadas) a:

$$
\begin{array}{l}
F[nil] \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1}:\text{entero})\ F[x_{k-1}::\ldots::x_1::nil] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k:\text{entero})\ F[x_k::x_{k-1}::\ldots::x_1::nil] \\
\quad \text{and} \\
(\forall\ x_1,\ldots,x_{k-1},x_k,x_{k+1}:\text{entero};l:\text{lista}) \\
\quad \left[\begin{array}{l} \text{if } F[x_k::\ldots::x_1::l] \\ \text{then } F[x_{k+1}::x_k::\ldots::x_1::l] \end{array}\right]
\end{array}
$$

que es el antecedente del principio de inducción con **k+1** casos básicos, como queríamos obtener. ∎

OTROS PRINCIPIOS DE INDUCCION ESTRUCTURAL

Podríamos desarrollar principios de inducción de manera similar a lo realizado con enteros y listas para otros tipos recursivos. Sólo incluimos, por completitud, el principio de

inducción estructural sobre conjuntos con dos casos, ya que se utiliza en la derivación desarrollada en el apartado 4.5.

```
    ⌈ F[φ]                                          ⌉
    |   and                                         |
if  | (∀ x:entero)                                   |
    | (∀ c:conjunto)  ⌈ if not x∈c                  ⌉|
    ⌊                 ⌊ then if F[c]                ⌋⌋
                              then F[x.c]
then (∀ c:conjunto) F[c]
```

El principio consta de las partes descritas en el apartado 2.4. El caso inductivo es ligeramente distinto de lo usual puesto que los conjuntos no se definen como un álgebra libre.

Debe destacarse que no siempre es fácil obtener principios alternativos de inducción estructural, y menos generalizar a un esquema general. Tipos de datos donde esto puede ser relativamente difícil son algunos árboles.

PRINCIPIO DE INDUCCION BIEN FUNDADA

La gran cantidad de principios de inducción concebibles puede resultar desconcertante. Sin embargo, existe un principio de inducción que generaliza todos los esquemas de inducción y es aplicable a cualquier tipo de datos recursivo. Se le conoce con el nombre de principio de inducción bien fundada (o completa). Por esta razón Manna y Waldinger lo han usado en su sistema de derivación de programas funcionales [MaWa80, MaWa87] como único principio de inducción.

El principio de inducción bien fundada se basa en el concepto de relación bien fundada (ver apartado A.6). Ejemplos de relaciones bien fundada son la relación menor-que $<$ sobre los enteros o la relación sublista $<_{\mathbf{lista}}$ sobre listas.

Podemos formular el principio de inducción de la siguiente forma. Supongamos que queremos demostrar que una sentencia **F[x]** es cierta para todo objeto **x** de tipo **T**. Basta probar que,

dado cualquier objeto **x**, si **F[x']** es cierto para todo objeto **x'** menor que **x** según algún orden bien fundado << (es decir, **x'<<x**), entonces **F[x]** también es cierto.

Formalmente, para cualquier sentencia **F[x]** sin apariciones libres de **x'**, el cierre universal de la sentencia

```
if (∀ x:T) [ if (∀ x':T) [ if x'<<x ]]
           [             [ then F[x'] ]]
           [ then F[x]                  ]
then (∀ x:T) F[x]
```

donde **x** no aparece libre en **F[x]**, es válida en la teoría del tipo **T**. Por brevedad en lo sucesivo no citaremos estas restricciones, que supondremos que se cumplen.

El principio de inducción bien fundada consta de partes similares a las contenidas en los principios de inducción estructural. La variable **x** se llama variable de inducción y la sentencia **F[x]** se llama la sentencia inductiva. El antecedente de la implicación se llama el paso inductivo. Asimismo, el antecedente y el consecuente del paso inductivo se llaman respectivamente la hipótesis de inducción y la conclusión deseada.

Si el tipo **T** es una tupla, la cuantificación se realiza sobre todos los elementos constituyentes de la tupla. Por ejemplo, para pares de listas, tendríamos:

```
if (∀ l1,l2:lista) [ if (∀ l3:lista) [ if (l3,l4)<<(l1,l2) ]]
                   [    (∀ l4:lista) [ then F[l3,l4]      ]]
                   [ then F[l1,l2]                          ]
then (∀ l1,l2:lista) F[l1,l2]
```

Los principios de inducción estructural son simplificaciones del principio de inducción bien fundada [MaWa89, cap.3] donde el orden bien fundado es "estructural", es decir, recoge la estructura recursiva de los datos. Por ejemplo, en la teoría de enteros el orden es $<_{\mathbf{pred}}$, definido como:

$$(\forall\ x,y{:}\text{entero})\ \left[\begin{array}{c} x <_{pred} y \\ \equiv \\ \text{succ } x = y \end{array}\right]$$

o en la teoría de listas el orden es $<_{cola}$, definido como:

$$(\forall\ l1,l2{:}\text{lista})\ \left[\begin{array}{c} l1 <_{cola} l2 \\ \equiv \\ (\exists\ x{:}\text{entero})\ x::l1 = l2 \end{array}\right]$$

A continuación utilizamos el principio de inducción bien fundada para obtener varios principios de inducción estructural sobre tuplas.

ORDENES BIEN FUNDADOS SOBRE TUPLAS

Podemos encontrar diversos principios de inducción estructural sobre tuplas de datos recursivos. Sus diferencias vienen del orden bien fundado << sobre tuplas elegido. Sólo vamos a utilizar dos órdenes bien fundados sobre tuplas, la relación lexicográfica y la relación progresiva.

Relación lexicográfica

Sea el tipo par $T_1 \# T_2$ y las relaciones bien fundadas $<_1$ y $<_2$ definidas respectivamente sobre los tipos T_1 y T_2. La relación lexicográfica $<_{lex}$ (correspondiente a $<_1$ y $<_2$, o simplemente correspondiente a $<_1$ si ambos órdenes son iguales) es un orden bien fundado [MaWa90, cap.4] que se define con el axioma:

$$\begin{array}{l}(\forall\ x_1,y_1{:}T_1) \\ (\forall\ x_2,y_2{:}T_2)\end{array}\ \left[\begin{array}{c} (x_1,x_2) <_{lex} (y_1,y_2) \\ \equiv \\ x_1 <_1 y_1 \\ \text{or} \\ x_1 = y_1 \quad \text{and} \quad x_2 <_2 y_2 \end{array}\right]$$

Obsérvese que en realidad la relación lexicográfica designa una familia de relaciones, que difieren en $<_1$ y $<_2$. La relación se define de manera similar para tuplas de longitud

mayor.

La relación lexicográfica es asimétrica en el peso dado para la comparación a cada elemento de las tuplas. Esta relación es la utilizada para la ordenación alfabética en los diccionarios: se comienza comparando los caracteres más a la izquierda de las dos palabras y sólo se continúa comparando los situados a la derecha si los anteriores son iguales.

Podemos formular restricciones de la relación lexicográfica que dan lugar a órdenes bien fundados simétricos, pero menos generales. Sólo estudiamos uno de ellos, el orden progresivo.

Relación progresiva

Sea el tipo par $T_1 \# T_2$ y las relaciones bien fundadas $<_1$ y $<_2$ definidas respectivamente sobre los tipos T_1 y T_2. La relación progresiva $<_{prog}$ (correspondiente a $<_1$ y $<_2$, o simplemente correspondiente a $<_1$ si ambos son iguales) es un orden bien fundado, que se define con el axioma:

$$(\forall\, x_1,y_1:T_1)(\forall\, x_2,y_2:T_2)\left[\begin{array}{l}(x_1,x_2) <_{prog} (y_1,y_2) \\ \equiv \\ x_1 <_1 y_1 \text{ and } x_2 = y_2 \\ \text{or} \\ x_1 = y_1 \text{ and } x_2 <_2 y_2\end{array}\right]$$

Al igual que la relación lexicográfica, la relación progresiva designa a una familia de relaciones, que difieren en $<_1$ y $<_2$. La relación se define de manera similar para tuplas de longitud mayor.

La relación progresiva es simétrica en el peso dado a cada elemento de una tupla en una comparación. Se llama progresiva porque la relación decrece si y sólo si decrece un elemento del par, pero a diferencia de la relación lexicográfica el otro elemento nunca puede aumentar simultáneamente.

Proposición (relación progresiva, bien fundada).

La relación progresiva es una relación bien fundada.

Demostración:

La relación progresiva es una subrelación de la relación lexicográfica, ya que si $t_1 <_{prog} t_2$, entonces $t_1 <_{lex} t_2$ (la inversa no es cierta en general). Además, la relación lexicográfica es una relación bien fundada. Por tanto, según la proposición de subrelación [MaWa90, cap.1] la relación progresiva es una relación bien fundada. ∎

PRINCIPIOS DE INDUCCION ESTRUCTURAL SOBRE PARES DE LISTAS

Una teoría de tuplas no suele incluir un principio de inducción estructural como axioma ya que una tupla no es un dato recursivo. Ahora bien, si los componentes de una tupla son recursivos, podemos razonar inductivamente mediante el principio de inducción bien fundada sobre tuplas. Si el orden bien fundado elegido depende del orden bien fundado estructural de los componentes recursivos de las tuplas, podemos formular el principio de inducción como una inducción estructural.

Encontramos dos principios ligeramente distintos de inducción estructural sobre pares de listas según utilicemos la relación lexicográfica o la relación progresiva. Dado que la segunda es una subrelación de la primera con una formulación muy parecida, no debe sorprender que los dos principios sean parecidos. El primero tiene un uso más general, mientras que el segundo es más sencillo de utilizar.

Inducción estructural por relación lexicográfica

El principio de inducción estructural sobre pares de lis-

tas por relación lexicográfica se formula:

$$
\begin{array}{l}
\text{if} \left[\begin{array}{l}
\left[\begin{array}{l} F[nil,nil] \\ \quad \text{and} \\ (\forall\ l_2\text{:lista}) \left[\begin{array}{l} \text{if } F[nil,l_2] \\ \text{then } F[nil,y::l_2] \end{array}\right] \\ (\forall\ y\text{:entero}) \end{array}\right] \\
\quad \text{and} \\
(\forall\ l_1\text{:lista};x\text{:entero}) \left[\begin{array}{l} \text{if } F[l_1,l_3] \\ \text{then } F[x::l_1,nil] \end{array}\right] \\
(\exists\ l_3\text{:lista}) \\
\quad \text{and} \\
(\forall\ l_1,l_2\text{:lista}) \left[\begin{array}{l} \text{if } \left[\begin{array}{l} F[l_1,l_3] \\ \quad \text{and} \\ F[x::l_1,l_2] \end{array}\right] \\ \text{then } F[x::l_1,y::l_2] \end{array}\right] \\
(\forall\ x,y\text{:entero}) \\
(\exists\ l_3\text{:lista})
\end{array}\right] \\
\text{then } (\forall\ l_1,l_2\text{:lista})\ F[l_1,l_2]
\end{array}
$$

Proposición (validez del principio de inducción estructural sobre pares de listas por relación lexicográfica).

El principio de inducción estructural sobre pares de listas por relación lexicográfica es válido en una teoría de pares de listas.

Demostración:

El principio de inducción bien fundada sobre tuplas es un principio válido para cualquier relación << bien fundada sobre tuplas, en concreto para la relación lexicográfica sobre pares correspondiente a $<_{\mathbf{cola}}$. Por tanto, basta determinar la equivalencia entre este principio y el principio de inducción estructural sobre listas de pares por relación lexicográfica para establecer la validez de este último principio. Ambos principios tienen el mismo consecuente, por lo que basta demostrar la equivalencia de los antecedentes.

Sean las definiciones de $<_{\mathbf{lex}}$ (para $<_{\mathbf{cola}}$) y de $<_{\mathbf{cola}}$:

$$
\begin{array}{l}
(\forall\ l_1,l_2\text{:lista}) \\
(\forall\ l_3,l_4\text{:lista})
\end{array}
\left[\begin{array}{c}
(l_3,l_4) <_{lex} (l_1,l_2) \\
\equiv \\
l_3 <_{cola} l_1 \\
\text{or} \\
l_3 = l_1 \quad \text{and} \quad l_4 <_{cola} l_2
\end{array}\right]
$$

$$(\forall\ l\text{:lista})\quad \text{not } l <_{cola} nil$$

$(\forall\ x{:}entero; l{:}lista)\quad l <_{cola} x::l$

Sea el antecedente del principio de inducción bien fundada sobre pares de listas con la relación lexicográfica antes mencionada.

$$(\forall\ l_1,l_2{:}lista)\ \left[\begin{array}{l}\text{if } (\forall\ l_3,l_4{:}lista)\ \left[\begin{array}{l}\text{if } (l_3,l_4)<_{lex}(l_1,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\ \text{then } F[l_1,l_2]\end{array}\right]$$

equivalente (por el teorema de descomposición de listas y lógica proposicional) a:

$$(\forall\ l_1,l_2{:}lista)\left[\begin{array}{l}\text{if } \left[\begin{array}{c} l_1=nil\\ \text{or}\\ (\exists\ x{:}entero; s{:}lista)\ l_1=x::s\end{array}\right]\\ \text{then } \left[\begin{array}{l}\text{if } (\forall\ l_3,l_4{:}lista)\\ \quad\left[\begin{array}{l}\text{if } (l_3,l_4)<_{lex}(l_1,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\ \text{then } F[l_1,l_2]\end{array}\right]\end{array}\right]$$

equivalente (por lógica proposicional y la proposición de reemplazamiento universal) a:

$$(\forall\ l_2{:}lista)\ \left[\begin{array}{l}\text{if } (\forall\ l_3,l_4{:}lista)\ \left[\begin{array}{l}\text{if } (l_3,l_4)<_{lex}(nil,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\ \text{then } F[nil,l_2]\end{array}\right]$$

and

$$(\forall\ l_2,s{:}lista)(\forall\ x{:}entero)\left[\begin{array}{l}\text{if } (\forall\ l_3,l_4{:}lista)\ \left[\begin{array}{l}\text{if } (l_3,l_4)<_{lex}(x::s,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\ \text{then } F[x::s,l_2]\end{array}\right]$$

equivalente (por el teorema de descomposición de listas y lógica proposicional) a:

$$(\forall\ l_2{:}lista)\left[\begin{array}{l}\text{if } l_2=nil \text{ then}\\ \text{if } (\forall\ l_3,l_4{:}lista)\ \left[\begin{array}{l}\text{if } (l_3,l_4)<_{lex}(nil,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\ \text{then } F[nil,l_2]\end{array}\right]$$

and

$$(\forall\ l_2,s'{:}lista)(\forall\ x'{:}entero)\left[\begin{array}{l}\text{if } l_2=x'::s' \text{ then}\\ \text{if } (\forall\ l_3,l_4{:}lista)\ \left[\begin{array}{l}\text{if } (l_3,l_4)<_{lex}(nil,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\ \text{then } F[nil,l_2]\end{array}\right]$$

and

$$
\begin{array}{l}
(\forall\ l_2,s{:}\mathrm{lista})\\
(\forall\ x{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } l_2=\mathrm{nil} \text{ then}\\
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\left[\begin{array}{l}\text{if } (l_3,l_4)<_{\mathrm{lex}}(x::s,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[x::s,l_2]
\end{array}\right]
$$

and

$$
\begin{array}{l}
(\forall\ l_2,s,s'{:}\mathrm{lista})\\
(\forall\ x,x'{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } l_2=x'::s' \text{ then}\\
\text{if}(\forall\ l_3,l_4{:}\mathrm{lista})\left[\begin{array}{l}\text{if } (l_3,l_4)<_{\mathrm{lex}}(x::s,l_2)\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[x::s,l_2]
\end{array}\right]
$$

equivalente (por la proposición de reemplazamiento universal) a:

$$
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\left[\begin{array}{l}\text{if } (l_3,l_4)<_{\mathrm{lex}}(\mathrm{nil},\mathrm{nil})\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[\mathrm{nil},\mathrm{nil}]
\end{array}\right]
$$

and

$$
\begin{array}{l}
(\forall\ s'{:}\mathrm{lista})\\
(\forall\ x'{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\left[\begin{array}{l}\text{if } (l_3,l_4)<_{\mathrm{lex}}(\mathrm{nil},x'::s')\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[\mathrm{nil},x'::s']
\end{array}\right]
$$

and

$$
\begin{array}{l}
(\forall\ s{:}\mathrm{lista})\\
(\forall\ x{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\left[\begin{array}{l}\text{if } (l_3,l_4)<_{\mathrm{lex}}(x::s,\mathrm{nil})\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[x::s,\mathrm{nil}]
\end{array}\right]
$$

and

$$
\begin{array}{l}
(\forall\ s,s'{:}\mathrm{lista})\\
(\forall\ x,x'{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\\
\quad\left[\begin{array}{l}\text{if } (l_3,l_4)<_{\mathrm{lex}}(x::s,x'::s')\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[x::s,x'::s']
\end{array}\right]
$$

equivalente (por las definiciones de $<_{\mathbf{lex}}$ y $<_{\mathbf{cola}}$ y la proposición de sustitutividad por equivalencia) a:

$$
F[\mathrm{nil},\mathrm{nil}]
$$

and

$$
\begin{array}{l}
(\forall\ s'{:}\mathrm{lista})\\
(\forall\ x'{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\\
\quad\left[\begin{array}{l}\text{if } l_3=\mathrm{nil} \text{ and } l_4<_{\mathrm{cola}}x'::s'\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[\mathrm{nil},x'::s']
\end{array}\right]
$$

and

$$
\begin{array}{l}
(\forall\ s{:}\mathrm{lista})\\
(\forall\ x{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\left[\begin{array}{l}\text{if } l_3<_{\mathrm{cola}}x::s\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[x::s,\mathrm{nil}]
\end{array}\right]
$$

and

$$
\begin{array}{l}
(\forall\ s,s'{:}\mathrm{lista})\\
(\forall\ x,x'{:}\mathrm{entero})
\end{array}
\left[\begin{array}{l}
\text{if } (\forall\ l_3,l_4{:}\mathrm{lista})\\
\quad\left[\begin{array}{l}\text{if } \left[\begin{array}{l}l_3<_{\mathrm{cola}}x::s\\ \quad\text{or}\\ l_3=x::s \text{ and } l_4<_{\mathrm{cola}}x'::s'\end{array}\right]\\ \text{then } F[l_3,l_4]\end{array}\right]\\
\text{then } F[x::s,x'::s']
\end{array}\right]
$$

equivalente (por la definición de $<_{cola}$ y la proposición de reemplazamiento universal) a:

$$
\begin{array}{l}
F[nil,nil] \\
\quad \text{and} \\
(\forall\ s':lista) \\
(\forall\ x':entero) \\
\quad \text{and} \\
(\forall\ s:lista; x:entero) \\
(\exists\ l_4:lista) \\
\quad \text{and} \\
(\forall\ s,s':lista) \\
(\forall\ x,x':entero) \\
(\exists\ l_4:lista)
\end{array}
\quad
\begin{array}{l}
\left[\begin{array}{l} \text{if } F[nil,s'] \\ \text{then } F[nil,x'::s'] \end{array}\right] \\
\left[\begin{array}{l} \text{if } F[s,l_4] \\ \text{then } F[x::s,nil] \end{array}\right] \\
\left[\begin{array}{l} \text{if } \left[\begin{array}{c} F[s,l_4] \\ \text{and} \\ F[x::s,s'] \end{array}\right] \\ \text{then } F[x::s,x'::s'] \end{array}\right]
\end{array}
$$

equivalente (por renombrando variables ligadas) a:

$$
\begin{array}{l}
F[nil,nil] \\
\quad \text{and} \\
(\forall\ l_2:lista) \\
(\forall\ y:entero) \\
\quad \text{and} \\
(\forall\ l_1:lista; x:entero) \\
(\exists\ l_3:lista) \\
\quad \text{and} \\
(\forall\ l_1,l_2:lista) \\
(\forall\ x,y:entero) \\
(\exists\ l_3:lista)
\end{array}
\quad
\begin{array}{l}
\left[\begin{array}{l} \text{if } F[nil,l_2] \\ \text{then } F[nil,y::l_2] \end{array}\right] \\
\left[\begin{array}{l} \text{if } F[l_1,l_3] \\ \text{then } F[x::l_1,nil] \end{array}\right] \\
\left[\begin{array}{l} \text{if } \left[\begin{array}{c} F[l_1,l_3] \\ \text{and} \\ F[x::l_1,l_2] \end{array}\right] \\ \text{then } F[x::l_1,y::l_3] \end{array}\right]
\end{array}
$$

que es el antecedente del principio de inducción estructural sobre pares de listas por relación lexicográfica, como queríamos obtener. ∎

Inducción estructural por relación progresiva

El principio de inducción estructural sobre pares de listas por relación progresiva se formula:

$$
\begin{array}{l}
\text{if} \left[\begin{array}{l}
F[nil,nil] \\
\quad \text{and} \\
(\forall\, l_2\text{:lista})\ \left[\begin{array}{l}\text{if } F[nil,l_2] \\ \text{then } F[nil,y::l_2]\end{array}\right] \\
(\forall\, y\text{:entero}) \\
\quad \text{and} \\
(\forall\, l_1\text{:lista})\ \left[\begin{array}{l}\text{if } F[l_1,nil] \\ \text{then } F[x::l_1,nil]\end{array}\right] \\
(\forall\, x\text{:entero}) \\
\quad \text{and} \\
(\forall\, l_1,l_2\text{:lista})\ \left[\begin{array}{l}\text{if } \left[\begin{array}{l}F[l_1,y::l_2] \\ \quad \text{and} \\ F[x::l_1,l_2]\end{array}\right] \\ \text{then } F[x::l_1,y::l_2]\end{array}\right] \\
(\forall\, x,y\text{:entero})
\end{array}\right] \\
\text{then } (\forall\, l_1,l_2\text{:lista})\ F[l_1,l_2]
\end{array}
$$

Proposición (validez del principio de inducción estructural sobre pares de listas por relación progresiva).

El principio de inducción estructural sobre pares de listas por relación progresiva es válido en una teoría de pares de listas.

Demostración:

La demostración es análoga a la realizada para la relación lexicográfica. ∎

PRINCIPIOS DE INDUCCION ESTRUCTURAL SOBRE PARES DE ENTEROS

De manera análoga a las listas, podemos formular dos principios de inducción sobre pares de enteros utilizando las relaciones lexicográfica y progresiva sobre pares de enteros.

Inducción estructural por relación lexicográfica

El principio de inducción estructural sobre pares de enteros por relación lexicográfica se formula:

```
    ⌈ F[0,0]                                                    ⌉
    |   and                                                     |
    | (∀ n':entero) ⌈ if F[0,n']      ⌉                         |
    |               ⌊ then F[0,n'+1] ⌋                          |
    |   and                                                     |
if  | (∀ n:entero)  ⌈ if F[n,m]       ⌉                         |
    | (∃ m:entero)  ⌊ then F[n+1,0] ⌋                           |
    |   and                                                     |
    |                                  ⌈   ⌈ F[n,m]    ⌉ ⌉      |
    | (∀ n,n':entero)  ⌈ if ⌈   and   ⌉ ⌉                       |
    | (∃ m:entero)     ⌊    ⌊ F[n+1,n'] ⌋ ⌋                     |
    ⌊                    ⌊ then F[n+1,n'+1] ⌋                   ⌋
then (∀ n,n':entero) F[n,n']
```

Inducción estructural por relación progresiva

Asimismo, el principio de inducción estructural sobre pares de enteros por relación progresiva se formula:

```
    ⌈ F[0,0]                                                    ⌉
    |   and                                                     |
    | (∀ n':entero) ⌈ if F[0,n']      ⌉                         |
    |               ⌊ then F[0,n'+1] ⌋                          |
    |   and                                                     |
if  | (∀ n:entero)  ⌈ if F[n,0]       ⌉                         |
    |               ⌊ then F[n+1,0] ⌋                           |
    |   and                                                     |
    |                                  ⌈   ⌈ F[n,n'+1] ⌉ ⌉      |
    | (∀ n,n':entero)  ⌈ if ⌈   and   ⌉ ⌉                       |
    |                  ⌊    ⌊ F[n+1,n'] ⌋ ⌋                     |
    ⌊                    ⌊ then F[n+1,n'+1] ⌋                   ⌋
then (∀ n,n':entero) F[n,n']
```

La justificación de ambos principios es análoga a la realizada para pares de listas, por lo que no la repetimos.

2.6. DEMOSTRACION POR DESCOMPOSICION

En el apartado anterior hemos utilizado diversos teoremas de descomposición para la deducción de los principios de inducción estructural. En este apartado estudiamos el uso de dichos teoremas para demostrar sentencias con cualquier formato, no solamente inductivo. Su uso nos lleva a la obtención de especificaciones parciales a partir de una especificación

completa. La demostración de todas aquéllas es equivalente a la demostración de esta última, pero su efecto derivador es distinto, como se ve en el Capítulo 3.

ESQUEMA DE DEMOSTRACION POR DESCOMPOSICION

Algunas sentencias cuantificadas universalmente sobre una variable se demuestran más fácilmente si se demuestra la misma sentencia para cada forma posible de construir el valor de la variable (según su tipo).

Proposición (equivalencia por descomposición).

Sea una declaración de tipo correspondiente a un álgebra libre:

data T == C_1 T_1 ++ ... ++ C_n T_n ;

Una sentencia con la forma:

(∀ x:T) F[x]

es equivalente a una sentencia con la forma:

(∀ $\bar{x}_1$:$\bar{T}_1$) F[C_1 s_1[$\bar{x}_1$]]
and
...
and
(∀ $\bar{x}_n$:$\bar{T}_n$) F[C_n s_n[$\bar{x}_n$]]

Demostración:

Dado que el tipo **T** es un álgebra libre, su teoría incluye, entre otros, un teorema de descomposición:

$$(\forall\ x{:}T)\ \left[\begin{array}{l}(\exists\ \bar{x}_1{:}\bar{T}_1)\ x = C_1\ s_1[\bar{x}_1] \\ \quad\text{or} \\ \quad\ldots \\ \quad\text{or} \\ (\exists\ \bar{x}_n{:}\bar{T}_n)\ x = C_n\ s_n[\bar{x}_n]\end{array}\right]$$

donde cada s_i ($1 \le i \le n$) es un término formado con constructores y las variables $\bar{x}_i$.

Sea la sentencia cuantificada original:

$$(\forall\ x{:}T)\ F[x]$$

equivalente (por el teorema de descomposición y lógica proposicional) a:

$$(\forall\ x{:}T)\ \left[\begin{array}{l}\text{if}\ \left[\begin{array}{l}(\exists\ \bar{x}_1{:}\bar{T}_1)\ x = C_1\ s_1[\bar{x}_1] \\ \quad\text{or} \\ \quad\ldots \\ \quad\text{or} \\ (\exists\ \bar{x}_n{:}\bar{T}_n)\ x = C_n\ s_n[\bar{x}_n]\end{array}\right] \\ \text{then}\ F[x]\end{array}\right]$$

equivalente (por lógica proposicional) a:

$$\begin{array}{l}(\forall\ x{:}T;\bar{x}_1{:}\bar{T}_1)\ \text{if}\ x = C_1\ s_1[\bar{x}_1]\ \text{then}\ F[x] \\ \quad\text{and} \\ \quad\ldots \\ \quad\text{and} \\ (\forall\ x{:}T;\bar{x}_n{:}\bar{T}_n)\ \text{if}\ x = C_n\ s_n[\bar{x}_n]\ \text{then}\ F[x]\end{array}$$

equivalente (por la proposición de reemplazamiento universal) a:

$$\begin{array}{l}(\forall\ \bar{x}_1{:}\bar{T}_1)\ F[C_1\ s_1[\bar{x}_1]] \\ \quad\text{and} \\ \quad\ldots \\ \quad\text{and} \\ (\forall\ \bar{x}_n{:}\bar{T}_n)\ F[C_n\ s_n[\bar{x}_n]]\end{array}$$

En particular, para los enteros una sentencia con la forma:

(∀ x:entero) F[x]

es equivalente a:

F[0]
  and
(∀ x:entero) F[succ x]

Asimismo, para las listas de enteros una sentencia con la forma:

(∀ l:lista) F[l]

es equivalente a la sentencia:

F[nil]
  and
(∀ x:entero;l:lista) F[x::l]

Observación (equivalencia por descomposición, alternativa).

La proposición de equivalencia por descomposición se refiere solamente a álgebras libres, o al menos a teorías que incluyen un teorema de descomposición con el formato indicado. Para otras teorías, la proposición debe reformularse en función de su teorema de descomposición.

Por ejemplo, la teoría de conjuntos incluye el siguiente teorema de descomposición:

$$(\forall\ s{:}\text{conjunto})\ \left[\begin{array}{c} s=\phi \\ \text{or} \\ (\exists\ x{:}\text{entero};s'{:}\text{conjunto}) \\ s=x.s'\ \ \text{and}\ \ \text{not}\ x\epsilon s' \end{array}\right]$$

que da lugar a que una sentencia con la forma:

(∀ s:conjunto) F[s]

sea equivalente a:

$$F[\phi]$$
$$\text{and}$$
$$(\forall\ x{:}\text{entero};s{:}\text{conjunto})\ \left[\begin{array}{l}\text{if not } x \epsilon s \\ \textit{then } F[x.s]\end{array}\right]$$

Su demostración es análoga a la de la proposición previa. ∎

Observación (demostración por inducción bien fundada y descomposición).

La sentencia F[x] puede tener cualquier formato, incluso ser una sentencia inductiva. Por ejemplo, supongamos que queremos demostrar una sentencia:

$$(\forall\ x{:}T)\ F[x]$$

Utilizando el principio de inducción bien fundada para un orden << sobre el tipo **T**, basta demostrar la sentencia:

$$(\forall\ x{:}T)\ \left[\begin{array}{l}\text{if } (\forall\ x'{:}T)\ \left[\begin{array}{l}\text{if } x' << x \\ \text{then } F[x']\end{array}\right] \\ \text{then } F[x]\end{array}\right]$$

Supongamos que el tipo **T** incluye en su teoría un teorema de descomposición con el formato deseado. Por la proposición de equivalencia por descomposición, basta demostrar la sentencia:

$$(\forall\ \bar{x}_1{:}\bar{T}_1)\ \left[\begin{array}{l}\text{if } (\forall\ x'{:}T)\ \left[\begin{array}{l}\text{if } x' << C_1\ s_1[\bar{x}_1] \\ \text{then } F[x']\end{array}\right] \\ \text{then } F[C_1\ s_1[\bar{x}_1]]\end{array}\right]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ \bar{x}_n{:}\bar{T}_n)\ \left[\begin{array}{l}\text{if } (\forall\ x'{:}T)\ \left[\begin{array}{l}\text{if } x' << C_n\ s_n[\bar{x}_n] \\ \text{then } F[x']\end{array}\right] \\ \text{then } F[C_n\ s_n[\bar{x}_n]]\end{array}\right]$$

y por tanto demostramos tantos conjuntados como constructores

definan el tipo **T**.

Para teorías que no son álgebras libres la descomposición tras inducción bien fundada puede formularse de otras maneras. Por ejemplo, puede probarse, utilizando la observación alternativa de equivalencia por descomposición, que para demostrar una sentencia:

(∀ s:conjunto) F[s]

en la teoría de conjuntos basta demostrar:

```
[ if (∀ s':conjunto) [ if s'<<φ ]]
[                    [ then F[s'] ]]
[ then F[φ]
    and
                [ if not xϵs
(∀ x:entero)    [ then [ if (∀ s':conjunto) [ if s'<<x.s ]]]
(∀ s:conjunto)  [      [                    [ then F[s'] ]]]]
                [      [ then F[x.s]
```

Observación (demostración por inducción estructural)

Supongamos que realizamos una demostración por inducción bien fundada y por descomposición, como en la observación anterior. Si la relación bien fundada es una relación estructural, la demostración resultante es una demostración por inducción estructural.

Veámoslo mejor con un caso concreto, cuando **T** es el tipo lista. La sentencia a demostrar es:

(∀ l:lista) F[l]

Por la observación anterior y la definición del tipo lista, basta demostrar la sentencia:

```
[ if (∀ l':lista) [ if l'<<nil ]]
[                 [ then F[l'] ]]
[ then F[nil]
   and
(∀ x:entero)  [ if (∀ l':lista) [ if l'<<x::l ]]
(∀ l:lista)   [                 [ then F[l'] ]]
              [ then F[x::l]
```

Tomando $<_{cola}$ como <<, definido por los axiomas:

(∀ l:lista) not $l <_{cola} nil$

(∀ x:entero;l:lista) $l <_{cola} x::l$

la sentencia anterior es equivalente a:

```
F[nil]
  and
(∀ x:entero)  [ if F[l]      ]
(∀ l:lista)   [ then F[x::l] ]
```

que es el antecedente del principio de inducción estructural sobre listas.

Podríamos prescindir de los principios de inducción estructural y deducirlos (mediante inducción bien fundada y descomposición) cuando se necesiten. Sin embargo los mantenemos por su mucha utilización. ∎

Debido al uso de la descomposición en la obtención de los principios de inducción estructural, en lo sucesivo hablamos de descomposición para referirnos tanto a su uso explícito (como ha ocurrido en la proposición de equivalencia por descomposición) como implícito (en la inducción estructural).

El razonamiento por descomposición puede generalizarse para sentencias cuantificadas universalmente sobre varias variables, de manera que se realice descomposición sobre todas ellas.

Proposición (equivalencia por descomposición, m variables).

Sean **m** declaraciones de tipos correspondientes a álgebras libres:

$$\text{data } T_1 == C_{11}\ T_{11} \mathbin{++} \ldots \mathbin{++} C_{1n1}\ T_{1n1}\ ;$$
$$\ldots$$
$$\text{data } T_m == C_{m1}\ T_{m1} \mathbin{++} \ldots \mathbin{++} C_{mnm}\ T_{mnm}\ ;$$

Una sentencia con la forma:

$$(\forall\ x_1{:}T_1;\ldots;x_m{:}T_m)\ F[x_1,\ldots,x_m]$$

es equivalente a una sentencia con la forma:

$$(\forall\ \bar{x}_{11}{:}\bar{T}_{11};\ldots;\bar{x}_{m1}{:}\bar{T}_{m1})\ F[C_{11}\ s_{11}[\bar{x}_{11}],\ldots,C_{m1}\ s_{m1}[\bar{x}_{m1}]]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ \bar{x}_{11}{:}\bar{T}_{11};\ldots;\bar{x}_{mnm}{:}\bar{T}_{mnm})\ F[C_{11}\ s_{11}[\bar{x}_{11}],\ldots,C_{mnm}\ s_{mnm}[\bar{x}_{mnm}]]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ \bar{x}_{1n1}{:}\bar{T}_{1n1};\ldots;\bar{x}_{m1}{:}\bar{T}_{m1})\ F[C_{1n1}\ s_{1n1}[\bar{x}_{1n1}],\ldots,C_{m1}\ s_{m1}[\bar{x}_{m1}]]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ \bar{x}_{1n1}{:}\bar{T}_{1n1};\ldots;\bar{x}_{mnm}{:}\bar{T}_{mnm})\ F[C_{1n1}\ s_{1n1}[\bar{x}_{1n1}],\ldots,C_{mnm}\ s_{mnm}[\bar{x}_{mnm}]]$$

Demostración:

La demostración es análoga a la realizada para una variable, sólo que utilizando **m** teoremas de descomposición, uno por variable a descomponer. ∎

Por ejemplo, una sentencia cuantificada sobre dos enteros con la forma:

$$(\forall\ x,x'\text{:entero})\ F[x,x']$$

es equivalente a:

$$F[0,0]$$
$$\text{and}$$
$$(\forall\ x\text{:entero})\ F[\text{succ } x,0]$$
$$\text{and}$$
$$(\forall\ x'\text{:entero})\ F[0,\text{succ } x']$$
$$\text{and}$$
$$(\forall\ x,x'\text{:entero})\ F[\text{succ } x,\text{succ } x']$$

DESCOMPOSICION DE UNA ESPECIFICACION

Supongamos que queremos demostrar una especificación de funciones con la forma usual:

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\ \ Q[\bar{x};\bar{z}]$$

o más explícitamente:

$$(\forall\ x_1:D_1;\ldots;x_i:D_i;\ldots;x_m:D_m)\ (\exists\ \bar{z}:\bar{R})$$
$$Q[(x_1,\ldots,x_i,\ldots,x_m);\bar{z}]$$

Si el tipo D_i de una variable x_i es un álgebra libre definido con n constructores, utilizando la proposición de equivalencia por descomposición, basta demostrar la sentencia:

$$(\forall\ x_1:D_1;\ldots;\bar{x}_{i1}:\bar{D}_{i1};\ldots;x_m:D_m)\ (\exists\ \bar{z}:\bar{R})$$
$$Q[(x_1,\ldots,s_{i1}[\bar{x}_{i1}],\ldots,x_m);\bar{z}]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ x_1;D_1;\ldots;\bar{x}_{ini}:\bar{D}_{ini};\ldots;x_m:D_m)\ (\exists\ \bar{z}:\bar{R})$$
$$Q[(x_1,\ldots,s_{ini}[\bar{x}_{ini}],\ldots,x_m);\bar{z}]$$

donde cada s_{ij} $(1\leq i\leq m,\ 1\leq j\leq n_i)$ es un término formado por las variables $\bar{x}_{ij}$ y constructores, o en forma simplificada:

$$(\forall\ \bar{x}_1{:}\bar{D}_1)\ (\exists\ \bar{z}{:}\bar{R})\quad Q[s_1[\bar{x}_1];\bar{z}]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ \bar{x}_n{:}\bar{D}_n)\ (\exists\ \bar{z}{:}\bar{R})\quad Q[s_{ni}[\bar{x}_{ni}];\bar{z}]$$

Pero esta sentencia es una conjunción de todas las especificaciones parciales de la función. Por tanto, para demostrar una especificación completa basta demostrar (por la semántica de la conectiva and) todas sus especificaciones parciales. Si la especificación contiene **m** variables de entrada y se demuestra por descomposición sobre varias variables, obtenemos una sentencia análoga a la anterior, con más conjuntados. Asimismo, si la especificación completa se demuestra por inducción bien fundada, todas las especificaciones parciales contienen un paso inductivo, como se ha visto en la observación de demostración por inducción bien fundada y descomposición. Por último, si la especificación completa se demuestra por inducción estructural, algunas de las especificaciones parciales contienen pasos inductivos, como se ha visto en la observación de demostración por inducción estructural. En resumen, el uso de la descomposición (explícita o implícita) para demostrar una especificación completa implica la demostración de todas sus especificaciones parciales, algunas quizá inductivamente.

3. SISTEMA DEDUCTIVO

Este capítulo desarrolla el sistema deductivo para la derivación de programas funcionales. El sistema es medianamente largo de describir; por esta razón comenzamos con una exposición informal, mediante un ejemplo, del papel de la deducción en la síntesis de funciones. Una vez adquirida cierta intuición del mecanismo deductivo, dedicamos tres apartados a la parte básica del mismo. Esta parte es prácticamente igual que la desarrollada por Manna y Waldinger, por lo que adoptamos su terminología y notación; también obliga a que su exposición sea concisa, con escasos ejemplos y demostraciones. (El lector puede consultar el Apéndice A, y para más detalle [MaWa85, MaWa86, MaWa87, MaWa89, MaWa90].) Posteriormente los apartados comprendidos entre el quinto y el octavo, ambos incluidos, amplían el cuadro con diversas facilidades de derivación de funciones con patrones. Esta parte adapta al cuadro deductivo los principios lógicos expuestos en el Capítulo 2. Por último, repasamos el proceso completo de derivación. Como parte de este proceso definimos un criterio para que las funciones auxiliares derivadas no utilicen parámetros de las funciones principales como variables globales.

3.1. DESCRIPCION INTUITIVA DE LA DERIVACION POR DEDUCCION

Ya hemos comentado en los apartados 1.2 y 2.3 que un sistema deductivo toma una sentencia lógica como especificación inicial y deriva un programa (funcional) durante su demostración. Veamos informalmente el proceso derivador de funciones con patrones con la ayuda de un ejemplo.

Supongamos que queremos derivar una función que calcule el máximo común divisor de dos enteros. En el apartado 4.1 realizamos una derivación parecida (la especificación cambia ligeramente) con el cuadro deductivo orientado a patrones. Una especificación semiformal de la función **mcd** es:

> dados dos enteros **i**, **j**, siendo alguno mayor que cero, s desea calcular una función **mcd** que calcule el máximo común divisor de **i** y **j**, es decir, que el máximo z tal que z|i y z|j (el símbolo | representa la relación "ser divisible entre" o "ser divisor de").

Esta especificación puede expresarse con la sentencia:

```
                  ┌ if i>0 or j>0                                    ┐
(∀ i,j:entero)    │                                                  │
(∃ z:entero)      │ then ┌ z|i and z|j and                          ┐ │
                  └      │ (∀ w:entero) ┌ if w|i and w|j            ┐ ┘
                         └              └ then w≤z                  ┘
```

La función a derivar tiene obviamente la siguiente declaración de tipo:

```
dec mcd : entero#entero -> entero ;
```

Si representamos la sentencia de especificación como:

```
(∀ i,j:entero) F[i,j]
```

el principio de inducción bien fundada sobre pares de enteros nos dice que para demostrar su validez basta con demostrar la validez de la sentencia:

```
(∀ i,j:entero) ┌ if (∀ u,v:entero) ┌ if (u,v)<<(i,j) ┐ ┐
               │                   └ then F[u,v]     ┘ │
               └ then F[i,j]                           ┘
```

Asimismo, podemos demostrar esta sentencia por descomposición de la variable entera **i**, lo que significa demostrar la sentencia:

```
(∀ j:entero) ┌ if (∀ u,v:entero) ┌ if (u,v)<<(0,j) ┐┐
             └ then F[0,j]       └ then F[u,v]     ┘┘
    and
(∀ i,j:entero)┌ if (∀ u,v:entero)┌ if (u,v)<<(succ i,j) ┐┐
              └ then F[succ i,j] └ then F[u,v]          ┘┘
```

La demostración de esta sentencia supone la demostración de sus dos conjuntados, que se hace por separado. Cada uno corresponde a una especificación parcial que va a originar una ecuación funcional.

La demostración de la primera especificación parcial no necesita de su hipótesis de inducción. Sea un entero **j** cualquiera. Debemos mostrar la validez de **F[0,j]**, es decir, suponemos que es cierto:

```
j>0
```

(ya que 0>0 es falso) y debemos encontrar un entero **z** para el que se cumpla:

```
z|0 and z|j and
(∀ w:entero) if w|0 and w|j then w≤z
```

Todo entero es divisible por 0, así que debe cumplirse:

```
z|j and (∀ w:entero) if w|j then w≤z
```

Esta sentencia se cumple si **z** toma el valor de **j** porque todo entero es divisible por sí mismo y cualquier entero divisible entre otro entero mayor que cero (**j** lo es por la asunción de entrada) es menor o igual que éste.

Por tanto hemos demostrado la primera especificación parcial obteniendo como ecuación:

```
--- mcd (0,j)
       <= j ;
```

La segunda especificación parcial sí va a necesitar la hipótesis de inducción. Sean dos enteros cualesquiera **i**, **j**. Suponemos que la hipótesis de inducción se cumple para cualesquiera enteros **u**, **v**; en la hipótesis la función **mcd** satisface la especificación. Es decir, asumimos la validez de la sentencia:

```
if (u,v)<<(succ i,j)
then [ if  u>0 or v>0
     [ then [ mcd(u,v)|u and mcd(u,v)|v  and
            [ (∀ w':entero) [ if w'|u and w'|v
                            [ then w'≤mcd(u,v) ]]]]
```

La condición de entrada parcial:

```
succ i>0 or j>0
```

es obviamente cierta, así que no necesitamos presuponer que se cumple.

Debemos encontrar un entero **z** que satisfaga:

```
z|succ i and z|j and
(∀ w:entero) if w|succ i and w|j then w≤z
```

Vamos a distinguir dos casos, cuando **succ i>j** y cuando **succ i≤j**. En el caso primero, si cambiamos el orden de los conjuntados, quedando:

```
z|j and z|succ i and
(∀ w:entero) if w|j and w|succ i then w≤z
```

la sentencia es satisfecha por la hipótesis de inducción, tomando **j**, **succ i**, **mcd(j,succ i)** por **u**, **v**, **z**. En consecuencia la ecuación actual tiene el siguiente formato:

```
--- mcd (succ i,j)
       <= if succ i>j
          then mcd(j,suc i)
          else ... ;
```

Por supuesto, para suponer que se cumple el consecuente de la hipótesis de inducción instanciada debemos demostrar que se cumplen sus dos antecedentes instanciados, cuya conjunción es:

```
(j,succ i)<<(succ i,j) and (j>0 or succ i>0)
```

Estos conjuntados sirven para garantizar respectivamente la condición de terminación y la condición de entrada en la recursión. El primer conjuntado se cumple tomando la relación lexicográfica $<_{lex}$ (sobre < y <) como << particular, ya que por la asunción de caso **j<succ i**. El segundo conjuntado también se cumple, porque su segundo disjuntado es obviamente cierto.

Veamos ahora el caso en que **succ i≤j**. La especificación parcial sigue siendo:

```
z|succ i and z|j and
(∀ w:entero) if w|succ i and w|j then w≤z
```

Si un entero es divisible entre dos enteros cualesquiera entonces también es divisible entre el menor de ambos (¡claro!) y el entero resultante de restar el menor del mayor. Utilizando esta propiedad y nuestra asunción de caso podemos reescribir la sentencia como:

```
z|succ i and z|j-succ i and
(∀ w:entero) if w|succ i and w|j-succ i then w≤z
```

Podemos satisfacer esta sentencia con la hipótesis de inducción, tomando **succ i**, **j-succ i**, **mcd(succ i,j-succ i)** por **u**, **v**, **z**. La rama "else" de la ecuación actual queda completa, resultando la ecuación:

```
--- mcd (succ i,j)
       <= if succ i>j
          then mcd(j,suc i)
          else mcd(succ i,j-succ i) ;
```

De nuevo deben verificarse las condiciones de terminación

y de entrada de la recursión, que instanciadas adecuadamente son:

$$(succ\ i, j-succ\ i) <_{lex(<,<)} (succ\ i, j) \text{ and } succ\ i>0 \text{ or } j-succ\ i>0$$

El primer conjuntado se satisface por la definición de la relación lexicográfica (sobre < y <) y por la asunción de caso. El primer conjuntado es obviamente cierto gracias a su primer disjuntado.

El programa completo derivado, tras introducir variables sinónimas y anónimas en la segunda ecuación, es:

```
dec mcd : entero#entero -> entero ;
--- mcd (0 , j)
       <= j ;
--- mcd (i & succ _ , j)
       <= if i>j then mcd (j,i) else mcd(i,j-i) ;
```

Resumiendo, hemos demostrado la validez de una sentencia de especificación por inducción bien fundada y descomposición sobre enteros. La información de la prueba la hemos utilizado para construir simultáneamente un programa funcional con patrones; de esta forma el programa satisface la especificación. El proceso utilizado, aunque sencillo e intuitivo, es informal y por tanto sujeto a errores. En el resto del capítulo desarrollamos un sistema formal para realizar derivaciones similares de programas.

3.2. SISTEMA DEDUCTIVO BASICO

El sistema deductivo es un sistema formal de demostración de teoremas con facilidades adicionales para la derivación de programas funcionales. En este apartado describimos el sistema básico de demostración de teoremas, posponiendo hasta el apartado 3.3 la exposición de la parte derivadora del sistema.

Como caso interesante para la derivación de funciones con patrones también examinamos la demostración de una conjunción de sentencias cerradas.

CUADRO DEDUCTIVO BASICO

La estructura fundamental del sistema deductivo es el cuadro ("tableau") deductivo [MaWa90,cap.11]. Un cuadro básico está formado por un conjunto de filas, cada una con dos columnas. Cada fila contiene una sentencia en una de las dos columnas; las sentencias de la primera columna se llaman asertos y las de la segunda columna, objetivos. Informalmente un aserto es una sentencia cuya verdad se supone y un objetivo es una sentencia cuya verdad se quiere demostrar.

Sea el cuadro siguiente:

asertos	objetivos
A_1	
...	
A_m	
	O_1
	...
	O_n

es deCir, sea un cuadro que contiene los asertos

$$A_1, \ldots, A_m$$

y los objetivos

$$O_1, \ldots, O_n.$$

Dicho cuadro representa la sentencia

if $(\forall^*)A_1$ and ... and $(\forall^*)A_m$
then $(\exists^*)O_1$ or ... or $(\exists^*)O_n$

que se llama su sentencia asociada. Asimismo, dicho cuadro es el cuadro asociado con esta sentencia. Dadas las propiedades de las conectivas **if-then, and** y **or**, el orden de las filas del cuadro es irrelevante.

Un cuadro (o su sentencia asociada) con asertos $A_1,\ldots,A_m$ y objetivos $O_1,\ldots,O_n$ es cierto bajo una interpretación I si se cumple la siguiente condición:

si los cierres universales $(\forall^*)A_i$ de todos los asertos A_i son ciertos bajo I,
entonces el cierre existencial $(\exists^*)O_j$ de algún objetivo O_j es cierto bajo I.

o equivalentemente, si:

el cierre universal $(\forall^*)A_i$ de algún aserto A_i
es falso bajo I
o
el cierre existencial $(\exists^*)O_j$ de algún objetivo O_j
es cierto bajo I.

Por tanto, un cuadro es obviamente cierto si uno de sus asertos es la sentencia **false** o uno de sus objetivos es **true**. Análogamente, el cuadro es falso bajo I si se cumple la siguiente condición:

los cierres universales $(\forall^*)A_i$ de todos los asertos A_i
son ciertos bajo I
y
los cierres existenciales $(\exists^*)O_j$ de todos los objetivos O_j
son falsos bajo I.

Un cuadro **C** es <u>válido</u> si, para cada interpretación **I**, **C** es cierto bajo **I**.

Dos cuadros **C** y **C'** son <u>equivalentes</u> si para cada interpretación **I**,

C es cierto bajo **I**
precisamente cuando
C' es cierto bajo **I**.

Se dice que dos cuadros **C** y **C'** tienen el <u>mismo significado</u> si:

C es válido
precisamente cuando
C' es válido.

Obsérvese que dos cuadros equivalentes también tienen el mismo significado, pero la inversa no tiene porqué cumplirse.

Polaridad y fuerza

Podemos extender la noción de <u>polaridad</u> de una sentencia (ver apartado A.1) a un cuadro. Dado un cuadro **C** y una subsentencia contenida en el mismo, asignamos una polaridad positiva (+), negativa (-) o ambas (±) a cada aparición de la subsentencia en el cuadro de acuerdo con las siguientes reglas de asignación de polaridad:

- Cada objetivo tiene una polaridad positiva en **C**.
- Cada aserto tiene una polaridad negativa en **C**.
- La polaridad de una subsentencia propia de un aserto u objetivo de **C** se determina según las reglas de asignación de polaridad a una subsentencia propia de una sentencia.

Una subsentencia de un cuadro tiene polaridad estricta en

el cuadro si no tiene ambas polaridades.

También puede extenderse la noción de fuerza de un cuantificador (ver apartado A.1) a un cuadro. Dado un cuadro **C** y una subsentencia **(...) F** del mismo, asignamos una fuerza universal (**∀**), existencial (**∃**) o ambas (**∀∃**) al cuantificador **(...)** de cada aparición de la subsentencia en el cuadro de la forma siguiente: se halla la polaridad de la subsentencia en el cuadro y entonces se aplican las reglas de asignación de fuerza a la subsentencia. Una aparición de un cuantificador en un cuadro tiene fuerza estricta en el cuadro si no tiene ambas fuerzas.

PROCESO DEDUCTIVO

En el sistema deductivo descrito, la validez de una sentencia **S** se determina hallando la validez de su cuadro asociado. El proceso es el siguiente:

- Formamos un cuadro inicial cuyo único objetivo es la sentencia **S**. También pueden incluirse como asertos cualesquiera sentencias válidas.
- Aplicamos sucesivamente reglas de deducción al cuadro; cada regla modifica el cuadro mediante la adición de filas nuevas. Las reglas de deducción conservan la validez del cuadro. En consecuencia, el cuadro resultante de aplicar una regla en cualquier etapa de la demostración tiene el mismo significado que el cuadro inicial.
- El proceso termina al obtener un cuadro obviamente válido, es decir, una fila con el aserto **false** o el objetivo **true**. Entonces se sabe que el cuadro inicial (y por tanto su sentencia asociada **S**) también es válido.

Se dice que el cuadro final es una prueba de la sentencia **S**.

Para derivar programas, la validez de las sentencias de especificación no se demuestra en la lógica de predicados pura, sino dentro de una teoría lógica. En este caso el cuadro inicial debe incluir como asertos los axiomas de la teoría; también pueden añadirse cualesquiera sentencias válidas en la teoría. Todas las teorías que usamos son teorías con igualdad, por lo que deben añadirse, entre otros, los axiomas de la igualdad. Si la teoría es una teoría con inducción, deben considerarse principios de inducción estructural o de inducción bien fundada. Sin embargo, para no hacer inmanejable el cuadro, supondremos que se han añadido al cuadro inicial los axiomas necesarios, aun sin escribirlos explícitamente.

Un problema con el esquema de sustitutividad de la igualdad y los principios de inducción es que son esquemas de axioma, y por tanto deberíamos incluir (según el vocabulario de la teoría) un número muy alto (quizá infinito) de axiomas en el cuadro. Un modo de evitar este inconveniente es definir reglas que tratan específicamente con la igualdad y la inducción. De este modo el único axioma que necesita seguir introduciéndose es el axioma de reflexividad de la igualdad. La descripción de dichas reglas se deja para más adelante.

SIMPLIFICACION

Cada vez que una fila se introduce en el cuadro es simplificada automáticamente lo máximo posible. Este proceso también afecta al cuadro inicial. Las simplificaciones facilitan el proceso de deducción, ya que hacen que las sentencias resultantes de aplicar las reglas de deducción sean más sencillas.

Una simplificación reemplaza una expresión (sentencia o término) por otra equivalente aunque más sencilla, es decir, más corta. Dado que la longitud de la expresión no puede acortarse indefinidamente, se garantiza que el proceso de simplificación termina alguna vez.

Una simplificación se representa de la forma:

```
E  =>  E'
```

donde **E** es una expresión sin simplificar y **E'** una expresión simplificada. **E** y **E'** son expresiones equivalentes, pero el reemplazamiento sólo se produce en el sentido de la flecha, es decir, podemos reemplazar todas las apariciones de una subsentencia de la forma **E** por las correspondientes sentencias de la forma **E'**, pero no a la inversa.

Existe un grupo de simplificaciones que eliminan apariciones de los símbolos **true** y **false** en una expresión. Por ejemplo, tenemos:

```
not true  =>  false
if F then false  =>  not F
F ≡ true  =>  F
```

Otro grupo de simplificaciones eliminan aplicaciones redundantes de las conectivas. Por ejemplo:

```
F and F  =>  F
not (F ≡ not G)  =>  F ≡ G
if F then t else t  =>  t
```

Por último, una regla permite eliminar igualdades (evidentes) de términos básicos:

```
t=t  =>  true
```

Por completitud, en el apartado A.2 se incluye el conjunto de simplificaciones utilizado.

REGLAS DE DEDUCCION

Podemos identificar siete grupos de reglas de deducción:

- Regla de reescritura, que reemplaza una subexpresión por otra expresión equivalente.
- Reglas de partición, que parten una fila en sus componentes lógicos.
- Regla de resolución, que realiza un análisis de casos sobre la verdad de una subsentencia.
- Regla de reemplazamiento por relación, que facilita el manejo de relaciones, como la equivalencia y la igualdad.
- Regla de unificación por relación, que permite la aplicación de cualquiera de las dos reglas anteriores mediante el establecimiento de cierta relación entre dos subexpresiones no unificables.
- Reglas de eliminación de cuantificadores (escolemización), que eliminan cuantificadores de las sentencias. Hay dos reglas, eliminación-$\forall$ y eliminación-$\exists$, que respectivamente eliminan los cuantificadores de fuerza universal y de fuerza existencial.

El sistema de Manna y Waldinger incluye una regla de inducción [MaWa89, MaWa90,cap.12]. Nosotros no la incluimos porque los razonamientos inductivos los haremos durante la construcción del cuadro deductivo, no sobre éste ya formado.

Todas las reglas anteriores conservan la equivalencia, excepto la regla de eliminación-$\forall$, que sólo conserva la validez.

La aplicación de una regla de deducción precisa que ciertas filas, llamadas las filas requeridas, estén presentes en el cuadro y hace que ciertas filas, llamadas las filas generadas, se introduzcan en el cuadro. En consecuencia podemos representar una regla en notación de cuadro de la forma:

	asertos	objetivos
C_r:		
C_g:		

donde las filas requeridas son aquéllas que aparecen sobre la línea doble y las filas generadas son aquéllas que aparecen bajo la línea doble.

Las filas requeridas forman el subcuadro requerido C_r, y las filas generadas forman el subcuadro generado C_g. El cuadro existente antes de aplicar una regla se llama el cuadro antiguo C_a. La aplicación de la regla produce la adición de las filas del subcuadro generado al cuadro antiguo, produciendo un cuadro nuevo C_n, es decir, $C_n = C_a \cup C_g$.

Las reglas de deducción deben ser consistentes. En lógica de predicados esto significa que deben conservar la validez del cuadro. Existen dos condiciones de justificación [MaWa90,cap.11] cuyo cumplimiento por una regla significa que la regla conserva respectivamente la validez y la equivalencia.

CUADRO DEDUCTIVO PARCIAL BASICO

Hemos visto cómo puede demostrarse una sentencia lógica en un cuadro deductivo. A veces conviene demostrar una sentencia por descomposición (explícita o implícita). Esto implica demostrar una conjunción de subsentencias en el cuadro deductivo; si la sentencia inicial es una especificación de programa, debe demostrarse la conjunción de sus especificaciones parciales. Una alternativa, que adoptamos por sus ventajas derivadoras, es demostrar todos los conjuntados, cada uno en un cuadro distinto. Cada cuadro se llama cuadro parcial; también llamaremos cuadro completo a un cuadro que no es parcial.

Proposición (demostración en cuadros parciales)

Sea la sentencia:

if $(\forall^*)A_1$ and ... and $(\forall^*)A_m$
then $(\exists^*)O_1$ and ... and $(\exists^*)O_n$

Resulta equivalente demostrar su validez en un cuadro deductivo completo que demostrar la validez de **n** sentencias

if $(\forall^*)A_1$ and ... and $(\forall^*)A_m$
then $(\exists^*)O_i$

para $1 \leq i \leq n$, cada una en un cuadro parcial.

Demostración:

Sea la sentencia original:

if $(\forall^*)A_1$ and ... and $(\forall^*)A_m$
then $(\exists^*)O_1$ and ... and $(\exists^*)O_n$

equivalente (por lógica proposicional) a:

if $(\forall^*)A_1$ and ... and $(\forall^*)A_m$
then $(\exists^*)O_1$
and
...
and
if $(\forall^*)A_1$ and ... and $(\forall^*)A_m$
then $(\exists^*)O_n$

Dado que esta sentencia es cerrada, también lo es cada conjuntado. Por tanto, la sentencia es válida si (por la semántica de la conectiva and) son válidos todos sus conjuntados. La validez de la sentencia original puede determinarse con un cuadro completo, mientras que la validez de cada conjuntado puede establecerse con un cuadro parcial. Como las dos sentencias son equivalentes, ambas formas de demostración son intercambiables. ∎

3.3. SISTEMA DEDUCTIVO AMPLIADO

Vemos ahora la ampliación del sistema deductivo básico para derivar términos funcionales. La mayor parte de la exposición está dedicada a justificar que los términos derivados son correctos, es decir, satisfacen la especificación demostrada, tanto si es completa como parcial.

CUADRO DEDUCTIVO AMPLIADO

Para la derivación de programas añadimos varias columnas, llamadas columnas de salida al cuadro deductivo. Un cuadro deductivo con columnas de salida se llama un cuadro ampliado.

Dada una especificación de programa

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\quad Q[\bar{x};\bar{z}]$$

o, más explícitamente:

$$(\forall\ x_1:D_1,\ldots,x_m:D_m)\ (\exists\ z_1:R_1,\ldots,z_n:R_n)\quad Q[(x_1,\ldots,x_m);z_1,\ldots,z_n]$$

con **n** variables de salida, el cuadro ampliado asociado tiene **n** columnas de salida. La columna j-ésima se utiliza para la derivación de la j-ésima función del programa. En cada fila del cuadro, las columnas de salida ora están todas en blanco ora cada una contiene algún término del lenguaje de programación funcional.

Las definiciones dadas en el apartado anterior sobre la verdad (bajo una interpretación) y la validez (en una teoría) de un cuadro siguen siendo aplicables al cuadro ampliado. Para dar significado a los términos de salida definimos lo que sig-

nifica que unos términos convengan ("suit") a un cuadro (bajo una interpretación) y que satisfagan un cuadro (en una teoría) [MaWa89]. De una manera informal, los términos que convienen a un cuadro denotan salidas aceptables para el programa deseado.

Definición (conveniencia y satisfacción entre términos y cuadro)

Sea un cuadro C, unos términos básicos $\bar{t} = t_1,\ldots,t_n$ y una interpretación I. Se dice que los términos $\bar{t}$ convienen al cuadro C bajo I si convienen a alguna fila de C. Los términos $\bar{t}$ convienen a una fila con una sentencia S si, para alguna sustitución θ, se cumplen las dos condiciones siguientes:

- condición de verdad: la sentencia $S\theta$ hace cierto el cuadro bajo I (es decir, es un aserto falso o un objetivo cierto).
- condición de salida: si la fila tiene unos términos de salida $\bar{s} = s_1,\ldots,s_n$, las instancias
  $$\bar{s}'\theta = s'_1\theta,\ldots,s'_n\theta$$
  de sus términos lógicos asociados $\bar{s}' = s'_1,\ldots,s'_n$ son básicas y tienen los mismos valores respectivos que $t_1,\ldots,t_n$ bajo I.

La sustitución θ se llama una sustitución adecuadora.

Unos términos básicos $\bar{t}$ satisfacen un cuadro C si convienen a C bajo cualquier modelo de la teoría. ∎

Observación (filas sin términos de salida)

En caso de que una fila no contenga términos de salida, la condición de salida obviamente se cumple, es decir, si la condición de verdad se cumple, cualesquiera términos $\bar{t}$ convienen a la fila. La misma situación se da cuando los términos de salida $\bar{s}$ son variables $\bar{u} = u_1,\ldots,u_n$, todas distintas y no

libres en la sentencia de la fila. En consecuencia podemos tanto añadir nuevas variables $\bar{u}$ en las columnas de salida si éstas están vacías como suprimirlas si no aparecen libres en la fila [MaWa89]. ∎

Las nociones de equivalencia y de mismo significado entre cuadros básicos se generalizan para cuadros ampliados. Dos cuadros **C** y **C'** son equivalentes en una teoría si, para cualquier modelo **I** de la teoría,

C es cierto bajo **I**
precisamente cuando
C' es cierto bajo **I**

y para cualesquiera términos básicos $\bar{t}$,

$\bar{t}$ convienen a **C** bajo **I**
precisamente cuando
$\bar{t}$ convienen a **C'** bajo **I**.

Análogamente, se dice que dos cuadros **C** y **C'** tienen el mismo significado en una teoría si

C es válido en la teoría
precisamente cuando
C' es válido en la teoría

y para cualesquiera términos básicos $\bar{t}$,

$\bar{t}$ satisface **C** en la teoría
precisamente cuando
$\bar{t}$ satisface **C'** en la teoría.

De nuevo, dos cuadros equivalentes tienen el mismo significado pero la inversa no tiene porqué cumplirse.

Propiedad (instanciación)

En una teoría, para cualquier sustitución **θ**,

un cuadro que contiene un aserto (u objetivo) **F** con términos de salida $\bar{s}$ (o ninguno)
es equivalente a
el cuadro que además contiene el aserto (u objetivo) **Fθ** con términos de salida $\bar{s}\theta$ (o ninguno). ∎

La propiedad del término de salida [MaWa89] relaciona términos de salida y especificaciones.

Propiedad (término de salida)

En cualquier teoría, si los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro:

asertos	objetivos	$\bar{f}$ $\bar{a}$
	$Q[\bar{a};\bar{z}]$	$\bar{z}$

donde $\bar{a}$ son constantes nuevas y $\bar{z}$ son las únicas variables libres en $Q[\bar{a};\bar{z}]$, entonces la sentencia

$$(\forall\ \bar{x}:\bar{D})\ Q[\bar{x};\bar{t}[\bar{x}]]$$

es válida en la teoría. ∎

PROCESO DERIVADOR

El proceso de derivar un programa que satisfaga una especificación $Q[\bar{x};\bar{z}]$ es similar al proceso de la demostración de ésta, ampliado con el tratamiento de las columnas de salida y la formación de las funciones derivadas:

- Dada la sentencia de especificación de **n** funciones f_i:

  $(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\ Q[\bar{x};\bar{z}]$

  tenemos directamente su declaración de tipo:

  dec f_1 : $\bar{D}$ -> R_1 ;
  ...
  dec f_n : $\bar{D}$ -> R_n ;

- Formamos un cuadro inicial cuya única fila es un objetivo con la sentencia $Q[\bar{x};\bar{z}]$ escolemizada y simplificada. La fila contiene en cada columna de salida i-ésima la variable de salida z_i. También se añaden como asertos los axiomas y cualesquiera sentencias válidas en la teoría; sus columnas de salida están vacías.

  En notación de cuadro, el cuadro inicial, con n+2 columnas, es:

asertos	objetivos	$\bar{f}$ $\bar{a}$
	Q[a;z]	$\bar{z}$

  más las filas necesarias para introducir como asertos los axiomas y otros teoremas de la teoría. En el cuadro los símbolos $\bar{a}$ son constantes nuevas (de Skolem) y $\bar{z}$ son variables, todos del tipo indicado en la especificación.

  Igual que para la mera demostración de teoremas, no es necesario que añadamos explícitamente los axiomas y teoremas de la teoría al cuadro, sino que suponemos que están contenidos en él.

- Aplicamos sucesivamente reglas de deducción al cuadro ampliado; cada regla modifica el cuadro mediante la adición de filas nuevas. Las reglas de deducción conservan

la validez del cuadro y los términos satisfactores del cuadro (es decir, un término básico satisfará el cuadro nuevo si y sólo si satisface el viejo).

- El proceso termina al obtener un cuadro obviamente válido. Es decir, la última fila del cuadro es:

false		$\bar{t}'[\bar{a}]$

o

	true	$\bar{t}'[\bar{a}]$

donde los términos $\bar{t}'[\bar{a}]$ deben ser primitivos. Estos términos pueden contener alguna variable, en cuyo caso la reemplazamos por una constante cualquiera, resultando unos términos $\bar{t}[\bar{a}]$.

Los términos lógicos correspondientes a $\bar{t}[\bar{a}]$ satisfacen claramente el cuadro final. Por tanto, también satisfacen el cuadro inicial, y el programa derivado:

$$\begin{aligned} &\text{--- f1 } \bar{x} \text{ <= } t_1[\bar{x}] \text{ ;} \\ &\qquad \cdots \\ &\text{--- fn } \bar{x} \text{ <= } t_n[\bar{x}] \text{ ;} \end{aligned}$$

satisface la especificación $Q[\bar{x};\bar{f}\ \bar{x}]$.

Cada función se forma fácilmente a partir del cuadro final, ya que es la expresión formada por el símbolo '---', el término cabecera de la función $f_i\ \bar{x}$, el símbolo '<=' el término de salida correspondiente $t_i[\bar{x}]$ y un punto y coma ';'.

Observación (primitividad)

Para que el programa derivado sea primitivo, se impone la restricción de que todos los términos de salida del cuadro sean términos primitivos. Por tanto, los únicos símbolos de Skolem que pueden aparecer son las constantes tomadas como parámetros. ∎

Ya sabemos que cada fila introducida en el cuadro es sometida automáticamente a un proceso de simplificación. En el sistema ampliado la simplificación afecta tanto a la sentencia como a los términos de salida.

Una regla de deducción en notación de cuadro consta de un cuadro requerido y un cuadro generado. En el sistema deductivo ampliado ambos cuadros deben incluir las columnas de asertos, objetivos y términos de salida.

Cada regla de deducción debe cumplir una condición de justificación ampliada, consistente en la condición de justificación que garantiza que la regla conserva la validez más una condición adicional [MaWa89] que garantiza que se mantienen los términos satisfactores.

El proceso derivador de un programa a partir de su especificación en una teoría se basa en una proposición sobre los términos de salida satisfactores del cuadro [MaWa89].

Proposición (términos de salida satisfactores)

Los sucesivos cuadros obtenidos durante la derivación cumplen que:

si los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro,
entonces la igualdad $\bar{f}\,\bar{x} = \bar{t}[\bar{x}]$ satisface la especificación. ∎

Lo expuesto hasta aquí es suficiente para derivar funciones mediante una sola ecuación. La derivación de funciones con varias ecuaciones exige una ligera generalización, que exponemos a continuación.

CUADRO DEDUCTIVO PARCIAL AMPLIADO

El proceso derivador expuesto se utiliza cuando las funciones se expresan con una sola ecuación. Si van a tener varias ecuaciones el proceso es ligeramente distinto. En líneas generales, la especificación se demuestra por descomposición (implícita o explícita), dando lugar a un conjunto de especificaciones parciales. Cada especificación parcial se demuestra en un cuadro parcial ampliado, derivando una ecuación por función. La justificación de este proceso requiere la reformulación de propiedades y proposiciones para especificaciones y cuadros parciales.

Una especificación parcial

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\quad Q[s[\bar{x}];\bar{z}]$$

o más explícitamente:

$$(\forall\ x_1:D_1;\ldots;\bar{x}_i:\bar{D}_i;\ldots;x_m:D_m)\ (\exists\ z_1:R_1;\ldots;z_n:R_n)\\ Q[(x_1,\ldots,t[\bar{x}_i],\ldots,x_m);z_1,\ldots,z_n]$$

tiene asociado un cuadro parcial ampliado con **n** columnas de salida. La columna j-ésima se utiliza para la derivación de una ecuación de la función j-ésima del programa, con cabecera $f_j(x_1,\ldots,t[\bar{x}_i],\ldots,x_m)$.

La propiedad modificada del término de salida relaciona términos de salida y especificaciones parciales.

Propiedad (término de salida parcial)

En cualquier teoría, si los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro parcial:

asertos	objetivos	$\bar{f}$ s[$\bar{a}$]
	Q[s[$\bar{a}$];$\bar{z}$]	$\bar{z}$

donde $\bar{a}$ son constantes nuevas y $\bar{z}$ son las únicas variables libres en $Q[s[\bar{a}];\bar{z}]$, entonces la sentencia

$$(\forall\ \bar{x}:\bar{D})\ Q[s[\bar{x}];\bar{t}[\bar{x}]]$$

es válida en la teoría.

Justificación:

Supongamos que los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro anterior. Para demostrar que $(\forall\ \bar{x}:\bar{D})\ Q[s[\bar{x}];\bar{t}[\bar{x}]]$ es válida (en la teoría), basta (por la proposición de eliminación de cuantificador universal) demostrar que:

$$Q[s[\bar{a}];\bar{t}[\bar{a}]]$$

es válida, ya que $\bar{a}$ son constantes nuevas. Sea un modelo cualquiera **I**; queremos mostrar que

$$Q[s[\bar{a}];\bar{t}[\bar{a}]]$$

es cierta bajo **I**.

Dado que los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro anterior, convienen a su única fila bajo **I**. Es decir, para alguna sustitución adecuadora **θ**, se cumplen la condición de verdad

$Q[s[\bar{a}];\bar{z}]\theta$ es cerrada y cierta bajo **I**,

y la condición de salida

$\bar{z}\theta$ son básicos y tienen los mismos valores respectivos que $\bar{t}[\bar{a}]$ bajo **I**.

Dado que $\bar{z}$ son todas las variables libres de $Q[s[\bar{a}];\bar{z}]$, esto significa que

$Q[s[\bar{a}];\bar{z}\theta]$ es cierta bajo **I**,

o equivalentemente (por la condición de salida)

$Q[s[\bar{a}];\bar{t}[\bar{a}]]$ es cierta bajo **I**,

como queríamos demostrar. ∎

El proceso derivador de una ecuación a partir de su especificación parcial se basa en modificaciones de las proposiciones de los términos de salida satisfactores para los sucesivos cuadros parciales del proceso derivador.

Proposición (cuadro parcial inicial)

En cualquier teoría,
si los términos básicos $\bar{t}[\bar{a}]$
satisfacen el cuadro inicial de una especificación parcial con parámetros $s[\bar{x}]$,
entonces la igualdad $\bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$
satisface la especificación parcial.

Demostración:

Supongamos que en una teoría los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro inicial para la especificación parcial

$Q[s[\bar{x}];\bar{z}]$. Debemos demostrar que la condición de corrección:

$$\text{if } (\forall\ \bar{x}{:}\bar{D})\ \ \bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$$
$$\text{then } (\forall\ \bar{x}{:}\bar{D})\ \ Q[s[\bar{x}];\bar{f}\ s[\bar{x}]]$$

es válida en la teoría.

Supongamos que (bajo cierto modelo)

$$(*)\ \ (\forall\ \bar{x}{:}\bar{D})\ \ \bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$$

es cierto. Debemos mostrar que

$$(\forall\ \bar{x}{:}\bar{D})\ \ Q[s[\bar{x}];\bar{f}\ s[\bar{x}]]$$

también es cierto.

Basta, dada nuestra asunción (*), demostrar que

$$(\forall\ \bar{x}{:}\bar{D})\ \ Q[s[\bar{x}];\bar{t}[\bar{x}]]$$

es cierto.

Por la propiedad del término de salida parcial, para demostrar que la sentencia anterior es válida, basta demostrar que los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro

asertos	objetivos	$\bar{f}\ s[\bar{a}]$
	$Q[s[\bar{a}];\bar{z}]$	$\bar{z}$

que hemos supuesto ser satisfecho. ∎

Proposición (cuadro parcial intermedio)

En cualquier teoría, la siguiente propiedad se cumple en

los sucesivos cuadros parciales obtenidos durante la derivación de una ecuación a partir de una especificación parcial:

si los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro parcial, entonces la igualdad $\bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$ satisface la especificación parcial.

Demostración:

La demostración se efectúa por inducción sobre las etapas de la demostración. Es decir, debemos demostrar que:

($) la propiedad se cumple para el cuadro inicial,

y

($$) si la propiedad se cumple para el cuadro antes de aplicar una regla de deducción, también se cumple después.

La condición ($) es justamente la proposición del cuadro parcial inicial, que es cierta.

Para demostrar ($$), suponemos que

(*) la propiedad se cumple para el cuadro **C** antes de aplicar una regla de deducción,

y debemos mostrar que la propiedad se cumple para el cuadro **C'** tras aplicar dicha regla. Para este fin, supongamos que

(**) los términos cerrados $\bar{t}[\bar{a}]$ satisfacen el cuadro **C'**

y mostramos que entonces la igualdad $\bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$ satisface su especificación parcial.

Puesto que las reglas de deducción mantienen los términos

satisfactores, sabemos que, dada nuestra asunción (**), $\bar{t}[\bar{a}]$ también satisfacen el cuadro C. Pero entonces, por nuestra asunción (*), la ecuación $\bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$ satisface la especificación parcial, como queríamos mostrar. ∎

Proposición (cuadro parcial final)

La igualdad $\bar{f}\ s[\bar{x}] = \bar{t}[\bar{x}]$ derivada de un cuadro parcial final satisface la especificación parcial $Q[s[\bar{x}];\bar{z}]$ en la teoría.

Demostración:

Por la proposición modificada del cuadro intermedio, basta demostrar que los términos básicos $\bar{t}[\bar{a}]$ satisfacen el cuadro parcial final en la teoría. Sea I cualquier modelo de la teoría; basta mostrar que $\bar{t}[\bar{a}]$ conviene al cuadro bajo I. En concreto mostraremos que $\bar{t}[\bar{a}]$ convienen a la fila final.

Recordemos que los términos lógicos $\bar{t}[\bar{a}]$ son unas instancias básicas de los términos lógicos $\bar{t}'[\bar{a}]$ correspondientes a los términos de salida de la fila final. Es decir, $\bar{t}[\bar{a}]$ son $\bar{t}'[\bar{a}]\theta$ para alguna sustitución θ. Tomemos θ como sustitución adecuadora.

Para demostrar la condición de verdad, debemos mostrar que la sentencia $(\mathbf{true})\theta$, es decir, **true**, es cierta y cerrada bajo I. Pero esto se cumple claramente.

Para demostrar la condición de salida, debemos mostrar que las instancias $\bar{t}'[\bar{a}]\theta$ son básicas y tienen los mismos valores respectivos que $\bar{t}[\bar{a}]$, pero esto se cumple por la elección de la sustitución adecuadora θ. ∎

En lo sucesivo no se escribirá el nombre de las columnas

salvo cuando queramos resaltar la cabecera $\tilde{f}\ s[\tilde{a}]$ de la ecuación que estamos derivando.

3.4. REGLAS DE DERIVACION

Exponemos con detalle las reglas de deducción del sistema, simplemente enunciadas en el apartado 3.2. Hemos esperado a formularlas hasta haber presentado el cuadro deductivo ampliado, ya que cada regla produce una sentencia deducida y un término de salida derivado.

REGLA DE REESCRITURA

La regla de reescritura permite reemplazar cierta subexpresión de un cuadro con una expresión equivalente o igual, según se trate respectivamente de una sentencia o un término. La regla añade una fila igual que la fila requerida, excepto en la expresión reemplazada por la reescritura en cuestión.

Una reescritura se representa de la forma:

E <=> E'

donde **E** y **E'** son dos expresiones equivalentes o iguales.

La reescritura se diferencia de la simplificación en que la sentencia introducida no tiene porqué ser más sencilla que la sentencia original. Además, la reescritura puede hacerse en cualquier sentido, es decir, podemos reemplazar una aparición de una subexpresión de la forma **E** por la correspondiente expresión de la forma **E'** o viceversa. Con cada aplicación de la regla sólo se reescribe una aparición de la expresión **E**.

Algunos ejemplos de reescrituras son:

```
if F then G else H <=> (if F then G) and (if not F then H)
(∀ x:T) [F and G]  <=>  (∀ x:T) F and (∀ x:T) G
p(x,if F then s else t,y)
     <=>  if F then p(x,s,y) else p(x,t,y)
```

Incluimos una reescritura, sólo aplicable a los términos de salida, que permite introducir definiciones locales:

$$t[t_1,\ldots,t_n] \;<=>\; \text{let } (x_1,\ldots,x_n) == (t_1,\ldots,t_n) \text{ in } t[x_1,\ldots,x_n]$$

donde $x_1,\ldots,x_n$ son variables que no aparecen libres en el término **t**.

En el caso particular de una sola variable local, la reescritura es:

$$t[t'] \;<=>\; \text{let } x == t' \text{ in } t[x]$$

Ejemplo (reescritura, raizcuad)

Sea la siguiente fila obtenida durante la derivación de una función que halla una aproximación inferior de la raíz cuadrada de un número **r** con margen de error **e**:

	$e \leq \max(r,1)$	if $(\text{raizcuad}(r,2*e)+e)^2 \leq r$ then raizcuad(r,2*e)+e else raizcuad(r,2*e)

Si aplicamos la reescritura de definiciones locales, tomando la llamada recursiva como término cualificado, queda:

	$e \leq \max(r,1)$	let v == raizcuad(r,2*e+e) in if $(v+e)^2 \leq r$ then v+e else v

El apartado A.2 incluye una relación de reescrituras.

REGLAS DE PARTICION

Las reglas de partición parten una fila en varias, conservando el sentido lógico de la sentencia asociada. Existen tres reglas de partición, todas muy intuitivas:

Regla de partición-and

A_1 and A_2		t
A_1		t
A_2		t

Regla de partición-or

	O_1 or O_2	t
	O_1	t
	O_2	t

Regla de partición-if

	if A then O	t
A		t
	O	t

Ejemplo (partición, mezclar)

Durante la derivación de un programa de ordenación por mezcla de listas se deriva una función auxiliar **mezclar** que, dadas dos listas ordenadas, mezcla sus elementos, produciendo una lista ordenada. La función **mezclar** puede derivarse por

inducción estructural sobre pares de listas por relación progresiva. El caso correspondiente a las dos listas no vacías (de cabecera **mezclar (x::l1,y::l2)**) tiene como cuadro parcial inicial:

	if [if ord l_1 and ord(y::l_2) then [perm (l_1<>y::l_2,mezclar(l_1,y::l_2)) and ord mezclar (l_1,y::l_2)] and if ord(x::l_1) and ord l_2 then [perm (x::l_1<>l_2,mezclar(x::l_1,l_2)) and ord mezclar(x::l_1,l_2)]] then [if ord(x::l_1) and ord(y::l_2) then perm (x::l_1<>y::l_2,z) and ord z]	z

Realizando dos veces partición-if resultan las tres nuevas filas:

if ord l_1 and ord(y::l_2) then [perm (l_1<>y::l_2,mezclar(l_1,y::l_2)) and ord mezclar (l_1,y::l_2)] and if ord(x::l_1) and ord l_2 then [perm (x::l_1<>l_2,mezclar(x::l_1,l_2)) and ord mezclar(x::l_1,l_2)]		
ord(x::l_1) and ord(y::l_2)		
	perm (x::l_1<>y::l_2,z) and ord z	z

Hemos suprimido la variable **z** de salida de la columna de salida de los dos asertos debido a que la variable no aparece libre en las sentencias de las filas (ver observación sobre filas sin términos de salida, apartado 2.3).

Si ahora aplicamos partición-and sobre los dos asertos obtenemos los nuevos asertos:

if ord(l_1 and ord($y::l_2$) then [perm (l_1<>$y::l_2$,mezclar(l_1,$y::l_2$)) and [ord mezclar (l_1,$y::l_2$)]		
if ord($x::l_1$) and ord l_2 then [perm ($x::l_1$<>l_2,mezclar($x::l_1$,l_2)) and [ord mezclar($x::l_1$,l_2)]		
ord($x::l_1$)		
ord($y::l_2$)		
	perm ($x::l_1$<>$y::l_2$,z) and ord z	z

Este proceso se realiza automáticamente al crear cada cuadro inicial, de modo que podíamos haberlo formado directamente con los cuatro últimos asertos más el objetivo previo. ■

REGLAS DE RESOLUCION

La regla de resolución realiza un análisis de casos sobre la verdad de una sentencia que aparece como una subsentencia común a dos filas. Dicha subsentencia puede no existir inicialmente pero formarse como consecuencia de una instanciación de las dos filas, hallada mediante unificación de las dos subsentencias.

Existen cuatro versiones distintas de la regla de resolución: resolución-AA (aplicada a dos asertos), resolución-AO (aplicada a un aserto y un objetivo), resolución-OA (aplicada a un objetivo y un aserto) y resolución-OO (aplicada a dos objetivos). Veamos primero la regla de resolución-AA, que suele considerarse la forma básica de la regla.

Forma básica

Dados dos asertos $\mathbf{A_1}$ y $\mathbf{A_2}$ estandarizados mutuamente, es decir, sin variables comunes:

$A_1[\bar{P}]$		$\bar{t}_1$
$A_2[\bar{P}']$		$\bar{t}_2$
$A_1\theta[\overline{\mathbf{false}}]$ or $A_2\theta[\overline{\mathbf{true}}]$		if $P\theta$ then $\bar{t}_1\theta$ else $\bar{t}_2\theta$

donde:

- $\bar{P}$ representa ciertas subsentencias libres y sin cuantificadores $P_1,\ldots,P_k$ (para $k\geq 1$) que aparecen en A_1,
- $\bar{P}'$ representa ciertas subsentencias libres y sin cuantificadores $P'_1,\ldots,P'_1$ (para $1\geq 1$) que aparecen en A_2,
- θ es un unificador más general de todas las subsentencias de $\bar{P}$ y $\bar{P}'$,
- $\overline{\mathbf{false}}$ representa **false,...,false** (**k** veces),
- $\overline{\mathbf{true}}$ representa **true,...,true** (**l** veces).

Se dice que el nuevo aserto es un resolvente, obtenido al aplicar la regla de resolución a los asertos $A_1[\bar{P}]$ y $A_2[\bar{P}']$, donde $\bar{P}$ y $\bar{P}'$ encajan mediante θ.

Si uno de los dos asertos carece de término de salida, el término derivado es simplemente el término del otro aserto (convenientemente instanciado). Si ninguno de los asertos tiene término de salida, el aserto derivado tampoco lo tiene. (Para su justificación, consultar [MaWa89].)

Veamos las tres formas restantes de la regla de resolución, que pueden considerarse variaciones de la forma básica.

Regla de resolución-AO

$A[\bar{P}]$		$\bar{t}_1$
	$O[\bar{P}']$	$\bar{t}_2$
	not $A\theta[\overline{false}]$ and $O\theta[\overline{true}]$	if $P\theta$ then $\bar{t}_1\theta$ else $\bar{t}_2\theta$

Regla de resolución-OA

	$O[\bar{P}]$	$\bar{t}_1$
$A[\bar{P}']$		$\bar{t}_2$
	$O\theta[\overline{false}]$ and not $A\theta[\overline{true}]$	if $P\theta$ then $\bar{t}_1\theta$ else $\bar{t}_2\theta$

Regla de resolución-OO

	$O_1[\bar{P}]$	$\bar{t}_1$
	$O_2[\bar{P}']$	$\bar{t}_2$
	$O_1\theta[\overline{false}]$ and $O_2\theta[\overline{true}]$	if $P\theta$ then $\bar{t}_1\theta$ else $\bar{t}_2\theta$

La formulación anterior de la regla de resolución se conoce como aclausal [MaWa80, Murray82] porque se diferencia de la tradicional formulación clausal [Robinson65] en que se aplica a subexpresiones (en vez de predicados) y que las sentencias no necesitan estar en forma normal alguna (p.ej. clausal). De esta manera se conserva la intuición sobre el proceso de derivación. Una ventaja adicional es que no tiene porqué aplicarse a dos filas diferentes, sino que puede aplicarse a una fila y ella misma.

La regla de resolución y las reglas de simplificación true-false forman un sistema completo para la lógica de predicados de primer orden [Murray82].

Restricción de polaridad

Hay varios modos de aplicar la regla de resolución entre dos filas, según la regla de resolución y las subsentencias $\bar{P}$ y $\bar{P}'$ elegidas. Algunas de estas aplicaciones no son útiles, pero podemos usar un criterio, llamado restricción de polaridad, para evitarlas.

Supongamos que $F_1[\bar{P}]$ y $F_2[\bar{P}']$ son dos asertos u objetivos de un cuadro con respectivas subsentencias libres y sin cuantificadores:

$$\bar{P} = P_1,\ldots,P_k \quad \text{y} \quad \bar{P}' = P'_1,\ldots,P'_l.$$

Supongamos también que la regla de resolución se aplica a $F_1[\bar{P}]$ y $F_2[\bar{P}']$, reemplazando cada aparición de $P\theta$ en $F_1\theta$ por **false** y cada aparición de $P\theta$ en $F_2\theta$ por **true**.

Entonces decimos que la regla se ha aplicado de acuerdo con la restricción de polaridad si

> al menos una de las subsentencias $P_1,\ldots,P_k$ de F_1 tiene polaridad negativa en el cuadro

y

> al menos una de las subsentencias $P'_1,\ldots,P'_l$ de F_2 tiene polaridad positiva en el cuadro.

Estas polaridades no tienen porqué ser estrictas.

Ejemplo (resolución, prodcart)

Supongamos que derivamos un programa, que halla el pro-

ducto cartesiano de dos conjuntos, por inducción estructural sobre el primer conjunto. En el cuadro parcial recursivo (de ecuación con cabecera **prodcart (x.s,t)**) obtenemos el objetivo:

	(a,b)ϵz ≡ (a,b)ϵprodlineal(x,t) or (a,b)ϵprodcart(s,t)	z

Recordemos que la propiedad **ϵ-U** de conjuntos es:

xϵsUt ≡ xϵs or sϵt		

Renombrando las variables **s, t** del aserto a **s', t'**, podemos resolver ambas filas con unificador más general {x<-(a,b), s'<-prodlineal(x,t), t'<-prodcart(s,t), z<-prodlineal(x,t)Uprodcart(s,t)}. El objetivo resultante es:

	true	prodlineal(x,t) U prodcart(s,t)

■

Ejemplo (resolución, raizcuad)

Sea el problema de hallar la mejor aproximación entera inferior a la raíz cuadrada de un entero **n** con margen de error **i**. El objetivo inicial es:

	$\boxed{z^2 \leq n}^+$ and not $(z+i)^2 \leq n$	z

Pongamos otra copia del objetivo, renombrando las variables:

	$z'^2 \leq n$ and not $\boxed{(z'+i)^2 \leq n}^-$	z'

Podemos resolver ambos objetivos (es decir, el objetivo inicial consigo) con un unificador {z<-z'+i}, quedando:

	$z'^2 \leq n$ and not $(z'+i+i)^2 \leq n$	if $(z'+i)^2 \leq n$ then z'+i else z'

Esta aplicación, poco usual, de la regla de resolución a un objetivo consigo mismo produce como término de salida una expresión condicional que contiene el esquema básico de búsqueda binaria en problemas numéricos [MaWa87]. ∎

Regla de resolución por teoría

La regla de resolución por teoría [Stickel85] es una extensión de la regla de resolución; aquí vemos una versión restringida [MaWa87, Traugott89] de la regla de Stickel. La nueva regla no añade potencia lógica al sistema, pero le permite realizar inferencias de una manera más eficiente. La regla se basa en suponer que ciertos teoremas de la teoría se encuentran incorporados al sistema sin necesidad de representarlos explícitamente como asertos. Esto evita muchas aplicaciones inútiles de la regla de resolución a dichos asertos, ya que sólo se invocan cuando se necesitan.

Supongamos que $H[\bar{P},\bar{Q}]$ es una sentencia válida que queremos incorporar. La regla de resolución por teoría, invocando el teorema $H[\bar{P},\bar{Q}]$, se formula:

	$O_1[\bar{P}']$	$\bar{t}_1$
	$O_2[\bar{Q}']$	$\bar{t}_2$
	$O_1\theta[\overline{true}]$ and $O_2\theta[\overline{true}]$ and not $H\theta[\overline{false},\overline{false}]$	if $P\theta$ then $\bar{t}_1\theta$ else $\bar{t}_2\theta$

donde θ es un unificador más general de $\bar{P}$, $\bar{P}'$ y de $\bar{Q}$, $\bar{Q}'$.

La restricción de polaridad obliga a que alguna sentencia de $\bar{P}'$ y $\bar{Q}'$ sea positiva en el cuadro y que alguna de $\bar{P}$ y $\bar{Q}$ lo sea en la sentencia **H**. Pueden describirse otras versiones de la regla para los casos en que alguna sentencia es un aserto o alguna subsentencia es negativa en el cuadro. En todo caso la polaridad de $\bar{P}$ en el cuadro tiene que tomarse contraria a la de $\bar{P}'$, sabiendo que **H** se introduce en el cuadro como un aserto. La misma restricción se aplica a $\bar{Q}$ y $\bar{Q}'$.

Ejemplo (resolución por teoría, insertar)

Supongamos que queremos derivar una función que, dados un entero y una lista de enteros ordenada en orden creciente, devuelve la lista ordenada crecientemente formada al insertar adecuadamente el entero en la lista inicial. Una derivación por inducción estructural permite obtener un cuadro parcial (para **insertar (x,y::l)**) con los dos objetivos siguientes:

	$\boxed{x \leq y}^{+}$	x::y::l
	$\boxed{y \leq x}^{+}$	y::insertar (x,l)

Estos objetivos no pueden reemplazarse directamente, pero sí puede hacerse invocando la propiedad **≤-totalidad**:

$\boxed{x' \leq y'}^{-}$ or $\boxed{y' \leq x'}^{-}$		

La resolución por teoría de las tres filas produce el objetivo final:

	true	if x≤y then x::y::l else y::insertar(x,l)

■

REGLA DE REEMPLAZAMIENTO POR RELACION

La regla de reemplazamiento por relación [MaWa86, Traugott89] permite reemplazar apariciones de subexpresiones por otras subexpresiones menores o mayores, según su polaridad, respecto a una relación <<. Para cada relación particular <<, esta regla la denominaremos regla de reemplazamiento por <<.

Para cualquier relación binaria <<, expresiones $\bar{I}$, $\bar{J}$, $\bar{I}'$ y $\bar{J}'$ y sentencias $o_1[\overline{\bar{I}<<\bar{J}}]$ y $o_2<\bar{I}'^{-<<},\bar{J}'^{+<<}>$, donde ambas sentencias están estandarizadas entre sí, tenemos:

	$o_1[\overline{\bar{I}<<\bar{J}}]$	$\bar{t}_1$
	$o_2<\bar{I}'^{-<<},\bar{J}'^{+<<}>$	$\bar{t}_2$
	$o_1\theta[\overline{false}]$ and $o_2\theta<\bar{J}\theta^{-<<},\bar{I}\theta^{+<<}>$	if $I\theta<<J\theta$ then $\bar{t}_2\theta$ else $\bar{t}_1\theta$

donde θ es un unificador simultáneo más general de $\bar{I}$, $\bar{I}'$ y $\bar{J}$, $\bar{J}'$.

Existe una restricción de polaridad que dice que la regla de reemplazamiento sólo se aplica si existe alguna aparición de $\overline{\bar{I}<<\bar{J}}$ negativa (o sin polaridad) en $o_1[\overline{\bar{I}<<\bar{J}}]$. También debe reemplazarse alguna aparición de $\bar{I}$ o de $\bar{J}$.

Existen dos instancias muy importantes de la regla de reemplazamiento por relación, obtenidas al tomar como relación << las relaciones de igualdad = y de equivalencia ≡. Se denominan **regla de reemplazamiento por igualdad** y **regla de reemplazamiento por equivalencia**, respectivamente. Su formulación es:

	$o_1[\overline{s=t}]$	$\bar{t}_1$
	$o_2\langle\bar{s}',\bar{t}'\rangle$	$\bar{t}_2$
	$o_1\theta[\overline{false}]$ and $o_2\theta\langle\bar{t}\theta,\bar{s}\theta\rangle$	if $s\theta=t\theta$ then $\bar{t}_2\theta$ else $\bar{t}_1\theta$

	$o_1[\overline{F\equiv G}]$	$\bar{t}_1$
	$o_2\langle\bar{F}',\bar{G}'\rangle$	$\bar{t}_2$
	$o_1\theta[\overline{false}]$ and $o_2\theta\langle\bar{G}\theta,\bar{F}\theta\rangle$	if $F\theta\equiv G\theta$ then $\bar{t}_2\theta$ else $\bar{t}_1\theta$

para términos $\bar{s}$, $\bar{t}$, $\bar{s}'$, $\bar{t}'$ y sentencias $\bar{F}$, $\bar{G}$, $\bar{F}'$, $\bar{G}'$, $o_1[\overline{s=t}]$, $o_2\langle\bar{s}',\bar{t}'\rangle$, $o_2\langle\bar{F}',\bar{G}'\rangle$. En estos dos casos no se distingue la polaridad de los términos reemplazados **sθ** y **tθ** (o **Fθ** y **Gθ**) en **Oθ** porque cualquier término tiene las dos polaridades respecto a la relación de igualdad o de equivalencia.

Ejemplo (reemplazamiento por igualdad, div-mod)

Sea una especificación de las funciones cociente y resto de dos enteros **i** y **j**. El objetivo inicial es:

	$i=\boxed{y*j}+z$ and $z<j$	y	z

donde **y**, **z** son las variables de salida de las funciones **div** y **mod**, respectivamente.

Utilizando el axioma ***-cero**,

$\boxed{0*x = 0}^{-}$			

y con un unificador {y<-0, x<-j}, podemos reemplazar por

igualdad en el objetivo inicial, obteniendo el nuevo objetivo:

	i=0+z and z<j	0	z

Ejemplo (reemplazamiento por permutación, ordrapida)

Supongamos que una especificación de ordenación de una lista se demuestra, según el esquema de ordenación rápida, mediante inducción bien fundada y descomposición en dos casos. La hipótesis de inducción para el caso de una lista no vacía es:

if l'<<x::l then [perm(l',ordrapida l')]$^{-}$ and ord ordrapida l'		

Sea el objetivo:

	perm (l, [ordrapida z_1] $^{\pm perm}$ <> ordrapida z_2) and if fϵ [ordrapida z_1] $^{\pm perm}$ then f$\leq$x and [if fϵ [ordrapida z_1] $^{\pm perm}$ and gϵordrapida z_2 then f$\leq$g] if gϵordrapida z_2 then x$\leq$g and z_1<<x::l and z_2<<x::l	ordrapida z_1 <> x::ordrapida z_2

El término **ordrapida w_1** tiene una polaridad positiva respecto a la relación de permutación. Por tanto podemos reemplazar, según la hipótesis de inducción, este término por w_1, resultando un objetivo:

	perm (l,z_1<>ordrapida z_2) and if fϵz_1 then f$\leq$x and if fϵz_1 and gϵordrapida z_2 then f$\leq$g and if gϵordrapida z_2 then x$\leq$g and z_1<<x::l and z_2<<x::l	ordrapida z_1 <> x::ordrapida z_2

REGLA DE UNIFICACION POR RELACION

A veces no puede aplicarse una regla de deducción porque la unificación falla en un solo término. Una solución es suponer que existe cierta relación entre ellos que permita realizar la unificación [MaWa86, Traugott89].

Sean los objetivos $O_1\langle J\rangle,\ldots,O_n\langle J\rangle$ y la regla de deducción **D**, y supongamos que la aplicación de esta regla a los objetivos produce el objetivo $D(O_1\langle J\rangle,\ldots,O_n\langle J\rangle)$. Entonces podemos aplicar **D** con unificación por la relación <<:

	$O_1\langle J\rangle$
	...
	$O_i\langle I^{-<<}\rangle$
	...
	$O_n\langle J\rangle$
	$I<<J$ and $D(O_1\langle J\rangle,\ldots,O_i\langle J^{-<<}\rangle,\ldots,O_n\langle J\rangle)$

La formulación anterior de la regla corresponde al caso en que no se necesita unificar los términos **J** de los objetivos. El término de salida y la táctica de polaridad son los correspondientes a la regla **D**.

Ejemplo (resolución con unificación por igualdad, menoresymayores)

Se quiere derivar una función que, dado un entero **x** y una lista **l** de enteros, devuelva una par de listas, formadas respectivamente por los elementos de **l** menores y mayores que **x**. Una demostración por inducción estructural sobre la lista de entrada nos da como objetivo inicial para el caso básico **meno-**

resymayores (x,nil):

	$\boxed{perm(nil, z_1 <> z_2)}^{+}$ and if $f \epsilon z_1$ then $f \leq x$ and if $g \epsilon z_2$ then $x \leq g$	(z_1, z_2)

El axioma **perm-nil** es:

$\boxed{perm\ (nil,nil)}^{-}$		

Directamente no podemos resolver ambas sentencias porque fracasamos al intentar unificar **nil** y $z_1 <> z_2$. Sin embargo utilizando unificación por igualdad la resolución puede efectuarse con éxito, produciendo el nuevo objetivo:

	$nil = z_1 <> z_2$ and if $f \epsilon z_1$ then $f \leq x$ and if $g \epsilon z_2$ then $x \leq g$	(z_1, z_2)

■

REGLAS DE ELIMINACION DE CUANTIFICADORES

Las reglas de eliminación de cuantificadores (o de escolemización) permiten eliminar cuantificadores de las sentencias de un cuadro. El valor principal de estas reglas es que preparan el cuadro para la aplicación de otras reglas (p.ej. resolución) que sólo se aplican a subsentencias libres y sin cuantificadores.

Hay dos reglas, la regla de eliminación-$\forall$, que elimina un cuantificador de fuerza universal estricta, y la regla de eliminación-$\exists$, que elimina un cuantificador de fuerza existencial estricta. Si el cuantificador tiene ambas fuerzas, debe estar dentro del ámbito de una conectiva de equivalencia $\equiv$ o dentro

de la cláusula **if** de una conectiva condicional o un término condicional. Un cuantificador de ambas fuerzas puede expresarse mediante varios cuantificadores de fuerza estricta aplicando reglas de reescritura.

La regla de eliminación-V no conserva la equivalencia del cuadro, sino solamente su validez, debido a la introducción de nuevos símbolos de constante o de función (llamados de Skolem).

Regla de eliminación-∀

A		**s**
A'		**s**

donde se cumplen las siguientes restricciones:

- El aserto requerido **A** contiene al menos una aparición de una subsentencia
  $(\ldots\ z{:}T)^{\forall}\ P[z]$
  donde $(\ldots\ \mathbf{z{:}T})^{\forall}$ tiene fuerza universal estricta en el cuadro; consideremos una aparición particular de esta subsentencia.
- Las únicas variables libres en **A** son $\bar{\mathbf{x}}$, es decir, $\mathbf{x_1},\ldots,\mathbf{x_m}$, donde $m\geq 0$.
- Los únicos cuantificadores de fuerza existencial que rodean la aparición de $(\ldots\ \mathbf{z{:}T})^{\forall}\ \mathbf{P[z]}$ son $(\ldots\ \bar{\mathbf{y}}{:}\bar{\mathbf{T}})^{\exists}$, es decir, $(\ldots\ \mathbf{y_1{:}T_1})^{\exists},\ldots,(\ldots\ \mathbf{y_n{:}T_n})^{\exists}$, donde $n\geq 0$.
- Las variables $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$, **z** son distintas; esto puede conseguirse si es necesario redenominando variables.
- El aserto generado **A'** se obtiene a partir de **A** reemplazando la aparición $(\ldots\ \mathbf{z{:}T})^{\forall}\ \mathbf{P[z]}$ por
  $P[f(\bar{x},\bar{y})]$
  donde **f** es un nuevo símbolo de función de Skolem.

En el caso especial en que no haya variables libres $\bar{\mathbf{x}}$ ni

cuantificadores $(\ldots\ \bar{y}{:}\bar{T})^{\exists}$, es decir, m=n=0, **A'** se obtiene de **A** reemplazando la aparición $(\ldots\ z{:}T)^{\forall}$ **P[z]** por

P[a]

donde **a** es un nuevo símbolo de constante de Skolem.

Los términos de salida permanecen igual.

Regla de eliminación-∃

A		s
A'		s

donde se cumplen las siguientes restricciones:

- El aserto requerido **A** contiene al menos una aparición de una subsentencia

  $(\ldots\ y{:}T)^{\exists}$ P[y]

  donde $(\ldots\ y{:}T)^{\exists}$ tiene fuerza existencial estricta en el cuadro; consideremos una aparición particular de esta subsentencia.
- La aparición de $(\ldots\ y{:}T)^{\exists}$ **P[y]** no está dentro del ámbito de ningún cuantificador de fuerza universal; esto puede conseguirse si es necesario aplicando previamente la regla de eliminación-∀.
- La aparición de $(\ldots\ y{:}T)^{\exists}$ **P[y]** no está dentro del ámbito de ningún otro cuantificador **(... y:T')** con la misma variable; esto puede conseguirse si es necesario redenominando variables.
- La variable **y** es distinta de cualquiera de las variables libres de **A**; esto puede conseguirse si es necesario redenominando variables.

El aserto generado **A'** se obtiene de **A** quitando la aparición del cuantificador $(\ldots\ y{:}T)^{\exists}$, es decir, reemplazando la aparición de $(\ldots\ y{:}T)^{\exists}$ **P[y]** por **P[y]**.

También existe la versión de ambas reglas para eliminar un cuantificador de un objetivo. Las restricciones para aplicar las reglas y el modo de obtener el objetivo generado son las mismas que para un aserto.

Generalmente no aplicaremos explícitamente las reglas anteriores, sino que se aplican automáticamente al crear el cuadro inicial.

3.5. DERIVACION DE FUNCIONES CON PATRONES

En los apartados anteriores hemos desarrollado los conceptos necesarios para derivar programas funcionales con patrones. Terminamos ahora de examinar el proceso derivador, justificando la derivación de varias ecuaciones a partir de una especificación completa; la idea central reside en demostrar la sentencia de especificación por descomposición. Esto significa demostrar las especificaciones parciales en cuadros parciales, derivando cAda uno una ecuación.

DEMOSTRACION PARCIAL SIN DERIVACION

Definición (especificación parcial obviamente válida)

Denominamos <u>especificación parcial obviamente válida</u> a una especificación parcial:

$$(\forall\ \bar{x}{:}\bar{D})\ (\exists\ \bar{z}{:}\bar{R})\ Q[s[\bar{x}];\bar{z}]$$

cuya demostración con un cuadro ampliado contiene una fila con aserto **false** (u objetivo **true**) sin términos en las columnas de salida. ∎

Ejemplo (especificación parcial obviamente válida, frenult)

Sea la especificación parcial de la función **frenult** para el caso en que el parámetro es la lista vacía:

```
(] z:entero#entero) [ if not nil=nil
                    [ then nil=primero z<>segundo z::nil ]
```

El cuadro parcial inicial correspondiente es:

asertos	objetivos	frenult nil
not nil=nil		
	nil=primero z<>segundo z::nil	z

Resolviendo el aserto inicial con al axioma de reflexividad de la igualdad, se obtiene el nuevo aserto:

false		

que no contiene ningún término de salida. Por tanto decimos que esta especificación parcial es obviamente válida. ∎

Según la observación de filas sin términos de salida (apartado 3.3), una fila sin término de salida puede tratarse como si tuviera términos $\bar{u} = u_1,\ldots,u_m$, siendo u_i variables libres distintas de las que aparecen en la fila. En nuestro caso no hay ninguna variable en la fila final, por lo que podemos incorporar variables $\bar{u}$. Ahora instanciamos las variables a constantes cualesquiera $\bar{c}$, obteniendo unos términos satisfactores $\bar{c}$. Por tanto, cada ecuación derivada es:

--- $f_i\ s[\bar{x}] <= c_i$;

Observación (contradicción de la condición de entrada)

Una especificación parcial obviamente válida es una especificación parcial cuya condición de entrada es falsa, es decir, cuyo parámetro toma un valor prohibido por la condición de entrada. Para comprobar esta afirmación basta examinar el proceso por el cual se deduce un aserto **false** (u objetivo **true**) sin término de salida.

La aplicación de una regla a filas donde alguna tiene término de salida produce filas nuevas con término de salida. Por tanto, un aserto (u objetivo) final sin términos de salida se obtiene tras una deducción en la que no interviene ninguna fila con término de salida. En el cuadro inicial la única fila con términos de salida es el objetivo inicial, así que la deducción se ha realizado a partir del resto de las filas, es decir, de los asertos.

La derivación de un aserto **false** a partir de otros asertos implica que se ha producido una contradicción [MaWa90,cap.6]. Una contradicción no puede producirse sólo a partir de los axiomas y teoremas de la teoría, ya que suponemos que ésta es consistente. Por tanto, algunos asertos del cuadro inicial deben ser producidos por partición del objetivo inicial de una especificación parcial. Estas asunciones sólo pueden ser condiciones de entrada, así que la contradición se ha producido por ser insatisfactible la condición de entrada correspondiente a la especificación parcial en cuestión. ∎

Una especificación parcial obviamente válida permite derivar una ecuación con un valor constante de salida cualquiera, como se vio antes de la observación. Sin embargo, puesto que representa aplicaciones de la función que no satisfacen la condición de entrada, no nos interesa el valor devuelto. Por tanto, adoptamos el criterio de no derivar ecuación alguna para estas especificaciones parciales.

Ejemplo (especificación parcial sin ecuación derivada, frenult)

La especificación parcial del ejemplo anterior contiene una condición de entrada que es falsa, produciendo una sentencia obviamente cierta. Dado que la condición de entrada no se cumple, no se deriva ninguna ecuación. ∎

DERIVACION DE FUNCIONES CON VARIAS ECUACIONES

Veamos la derivación de varias ecuaciones por función. Básicamente se trata de demostrar una sentencia de especificación por descomposición. Esto significa demostrar varias especificaciones parciales, cada una en un cuadro parcial.

Propiedad (derivación de funciones por descomposición)

Sea una especificación de unas funciones $\bar{f}$:

$$(\forall\ x_1{:}D_1,\ldots x_i{:}D_i,\ldots,x_m{:}D_m)\ (\exists\ \bar{z}{:}\bar{R})\ Q[(x_1,\ldots,x_i,\ldots,x_m);\bar{z}]$$

Si la teoría del tipo D_i contiene un teorema de descomposición:

$$(\forall\ x_i{:}D_i)\left[\begin{array}{l}(\exists\ \bar{x}_{i1}{:}\bar{T}_{i1})\ x = C_{i1}\ s_{i1}[\bar{x}_{i1}]\\ \qquad\qquad\qquad\qquad \text{or}\\ \qquad\qquad\qquad\qquad \ldots\\ \qquad\qquad\qquad\qquad \text{or}\\ (\exists\ \bar{x}_{ini}{:}\bar{T}_{ini})\ x = C_{ini}\ s_{ini}[\bar{x}_{ini}]\end{array}\right]$$

las funciones $\bar{f}$ se derivan construyendo n_i cuadros parciales, cada uno con objetivo inicial:

asertos	objetivos	$\bar{f}\ s_j[\bar{a}]$
	$Q[s_j[\bar{a}];\bar{z}]$	$\bar{z}$

donde $s_j[\bar{a}]$ representa el término
$(a_1,\ldots,c_{ij}\ s_{ij}[\bar{a}_{ij}],\ldots,a_m)$, para $1\leq j\leq n_i$.

Justificación:

Según se vio en el apartado 2.6, para demostrar la especificación completa basta demostrar la sentencia:

$$
\begin{array}{l}
(\forall\ x_1{:}D_1;\ldots;\bar{x}_{i1}{:}\bar{D}_{i1};\ldots;x_m{:}D_m)\ (\exists\ \bar{z}{:}\bar{R}) \\
\quad Q[(x_1,\ldots,s_{i1}[\bar{x}_{i1}],\ldots,x_m);\bar{z}] \\
\quad \text{and} \\
\quad \ldots \\
\quad \text{and} \\
(\forall\ x_1{:}D_1;\ldots;\bar{x}_{ini}{:}\bar{D}_{ini};\ldots;x_m{:}D_m)\ (\exists\ \bar{z}{:}\bar{R}) \\
\quad Q[(x_1,\ldots,s_{ini}[\bar{x}_{ini}],\ldots,x_m);\bar{z}]
\end{array}
$$

que es una conjunción de especificaciones parciales. Cada una es una sentencia cerrada, así que (por la proposición de demostración en cuadros parciales) demostramos cada conjuntado en un cuadro parcial. El cuadro parcial asociado a la especificación parcial:

$$
\begin{array}{l}
(\forall\ x_1{:}D_1;\ldots;\bar{x}_{ij}{:}\bar{D}_{ij};\ldots;x_m)\ (\exists\ \bar{z}{:}\bar{R}) \\
\quad Q[(x_1,\ldots,\bar{x}_{ij},\ldots,x_m);\bar{z}]
\end{array}
$$

para $1\leq j\leq n_i$ es:

asertos	objetivos	$\bar{f}\ s_j[\bar{a}_j]$
	$Q[s_j[\bar{a}_j];\bar{z}]$	$\bar{z}$

donde $s_j[\bar{a}_j]$ representa la tupla $(a_1,\ldots,s_{ij}[\bar{a}_{ij}],\ldots,a_m)$. ∎

La propiedad de descomposición se cumple igualmente si la descomposición se hace simultáneamente sobre **k** variables de entrada ($1\leq k\leq m$).

Proposición (corrección de programa derivado por descomposición)

En cualquier teoría,

si los términos básicos $\bar{t}_j[\bar{a}_j]$ satisfacen el cuadro parcial inicial de cabecera $\bar{f}\ s_j[\bar{a}_j]$ $(1 \le j \le n_i)$

y la teoría del tipo D_i contiene un teorema de descomposición en n_i casos $(1 \le i \le m)$,

entonces el programa de igualdades $\bar{f}\ s_j[\bar{a}_j] = \bar{t}_j[\bar{a}_j]$ satisface la especificación.

Demostración:

Para demostrar que el programa de igualdades $\bar{f}\ s_j[\bar{a}_j] = \bar{t}_j[\bar{a}_j]$ satisface la especificación, debemos demostrar la validez de la condición de corrección:

$$\text{if} \left[\begin{array}{l} (\forall\, \bar{x}_1 : \bar{D}_1)\ \ \bar{f}\ s_1[\bar{x}_1] = \bar{t}_1[\bar{x}_1] \\ \qquad\qquad \text{and} \\ \qquad\qquad \ldots \\ \qquad\qquad \text{and} \\ (\forall\, \bar{x}_{ni} : \bar{D}_{ni})\ \ \bar{f}\ s_{ni}[\bar{x}_{ni}] = \bar{t}_{ni}[\bar{x}_{ni}] \end{array}\right]$$
$$\text{then}\ \ (\forall\, \bar{x} : \bar{D})\ \ Q[\bar{x}; \bar{f}\ \bar{x}]$$

Bajo cierto modelo de la teoría supongamos que cada conjuntado del antecedente

$$(*)\ \ (\forall\, \bar{x}_j : \bar{D}_j)\ \ \bar{f}\ s_j[\bar{x}_j] = \bar{t}_j[\bar{x}_j]$$

es cierto y debemos mostrar que el consecuente:

$$(\forall\, \bar{x} : \bar{D})\ \ Q[\bar{x}; \bar{f}\ \bar{x}]$$

también es cierto.

Como la teoría del tipo D_i contiene un teorema de descomposición, basta (por la proposición de equivalencia por descomposición, apartado 2.6) demostrar la sentencia:

$$(\forall\ \bar{x}_1{:}\bar{D}_1)\ \ Q[s_1[\bar{x}_1];\bar{f}\ s_1[\bar{x}_1]]$$
$$\text{and}$$
$$\ldots$$
$$\text{and}$$
$$(\forall\ \bar{x}_{ni}{:}\bar{D}_{ni})\ \ Q[s_{ni}[\bar{x}_{ni}];\bar{f}\ s_{ni}[\bar{x}_{ni}]]$$

Por nuestras asunciones (*), 1 de los conjuntados ($1 \leq l \leq m$) pueden expresarse como:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \ Q[s_j[\bar{x}_j];\bar{t}_j[\bar{x}_j]]$$

Para los casos restantes, las funciones $\bar{f}$ no están definidas, por lo que podemos asociarles una valor constante c_j cualquiera y expresarlos:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \ Q[s_j[\bar{x}_j];\bar{c}_j]$$

La sentencia resultante es cierta (por la semántica de la conectiva and) si lo es cada uno de sus conjuntados. Según la propiedad del término de salida parcial, cada conjuntado con la forma:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \ Q[s_j[\bar{x}_j];\bar{t}_j[\bar{x}_j]]$$

es cierto si los términos $\bar{t}_j[\bar{x}_j]$ satisfacen el cuadro inicial de la especificación parcial:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ (\exists\ \bar{z}{:}\bar{R})\ \ Q[s_j[\bar{x}_j];\bar{z}]$$

como suponemos que ocurre.

Cada conjuntado con la forma:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \ Q[s_j[\bar{x}_j];\bar{c}_j]$$

es cierto si los términos $\bar{c}_j$ satisfacen el cuadro inicial de la especificación parcial correspondiente. Pero como es una especificación parcial obviamente válida, cualesquier términos básicos $\bar{c}_j$ satisfacen su cuadro inicial. ∎

Observación (asertos obvios en cuadros parciales)

Los cuadros parciales creados durante una demostración por descomposición se parten automáticamente, como siempre. Como consecuencia a veces contienen asertos que son obviamente ciertos porque son instancias de axiomas o teoremas de la teoría. Por la propiedad de instanciación de los cuadros deductivos ampliados (apartado 2.3) podemos eliminar estos asertos, obteniendo un cuadro parcial inicial equivalente pero más sencillo. ∎

3.6. DERIVACION DE FUNCIONES RECURSIVAS

Presentamos en este apartado los mecanismos de derivación de funciones recursivas. La derivación de una función recursiva implica el uso de algún principio de inducción, como se razonó en el apartado 2.4. Distinguimos el uso de distintos principios de inducción, ya que tienen efectos diferentes sobre el cuadro deductivo, y por tanto sobre el programa derivado. Así, el uso de inducción estructural provoca la creación de varios cuadros parciales, mientras que el uso de inducción bien fundada no.

USO DE INDUCCION BIEN FUNDADA

Recordemos el principio de inducción bien fundada sobre tuplas, formulada:

$$\text{if } (\forall\, \bar{x}{:}\bar{T})\ \left[\begin{array}{l}\text{if } (\forall\, \bar{x}'{:}\bar{T})\ \left[\begin{array}{l}\text{if } \bar{x}'<<\bar{x}\\ \text{then } F[\bar{x}']\end{array}\right]\\ \text{then } F[\bar{x}]\end{array}\right]$$
$$\text{then } (\forall\, \bar{x}{:}\bar{T})\ F[\bar{x}]$$

Supongamos que disponemos de una especificación completa:

$(\forall\ \bar{x}{:}\bar{D})\ (\exists\ \bar{z}{:}\bar{R})\ Q[\bar{x};\bar{z}]$

Esta sentencia de especificación tiene el mismo formato que el consecuente del principio de inducción bien fundada, tomando respectivamente $\bar{D}$ y $(\exists\ \bar{z}{:}\bar{R})\ Q[\bar{x};\bar{z}]$ por $\bar{T}$ y $F[\bar{x}]$. Por tanto, para demostrar su validez (en la teoría) basta demostrar la validez del antecedente del principio instanciado.

Definición (cuadro inicial recursivo)

Dadas, en una teoría, una especificación:

$(\forall\ \bar{x}{:}\bar{D})\ (\exists\ \bar{z}{:}\bar{R})\ \ Q[\bar{x};\bar{z}]$

y una relación << bien fundada definida sobre tuplas de tipo D, un cuadro inicial recursivo se define como:

asertos	objetivos	$\bar{f}\ \bar{a}$
if $\bar{a}'<<\bar{a}$ then if $E[\bar{a}']$ then $S[\bar{a}';\bar{f}\ \bar{a}']$		
$E[\bar{a}]$		
	$S[\bar{a};\bar{z}]$	$\bar{z}$

■

La justificación de la introducción del término $\bar{f}\ \bar{a}'$ en la hipótesis de inducción y la demostración de corrección del programa recursivo derivado pueden encontrarse en [MaWa89].

Observación (demostración por inducción bien fundada y descomposición)

Recordemos (apartado 2.6) que una sentencia puede demostrarse combinando los principios de inducción bien fundada y

descomposición. En consecuencia para demostrar una sentencia con la forma:

$$(\forall\ \bar{x}:\bar{T})\ F[\bar{x}]$$

basta demostrar una sentencia con la forma:

$$\begin{array}{l}
(\forall\ \bar{x}_1:\bar{T}_1)\ \left[\begin{array}{l}\text{if } (\forall\ \bar{x}':\bar{T})\ \left[\begin{array}{l}\text{if } \bar{x}'<<C_1\ s[\bar{x}_1]\\ \text{then } F[\bar{x}']\end{array}\right]\\ \text{then } F[C_1\ s_1[\bar{x}_1]]\end{array}\right]\\
\quad\text{and}\\
\quad\ldots\\
\quad\text{and}\\
(\forall\ \bar{x}_n:\bar{T}_n)\ \left[\begin{array}{l}\text{if } (\forall\ \bar{x}':\bar{T})\ \left[\begin{array}{l}\text{if } \bar{x}'<<C_n\ s_n[\bar{x}_n]\\ \text{then } F[\bar{x}']\end{array}\right]\\ \text{then } F[C_n\ s_n[\bar{x}_n]]\end{array}\right]
\end{array}$$

Por tanto, si la sentencia por demostrar es una especificación completa, basta demostrar por inducción bien fundada las **n** especificaciones parciales mediante **n** cuadros parciales recursivos. Cada cuadro parcial recursivo se construye análogamente al cuadro recursivo previo; su partición y la eliminación de asertos que son instancias de sentencias válidas se hace automáticamente. Si una especificación inductiva parcial es obviamente válida su cuadro parcial no deriva ninguna ecuación.

La justificación de todo el proceso es una mezcla de la justificación de la derivación de funciones por descomposición y de los cuadros iniciales recursivos. ∎

Ejemplo (cuadros iniciales recursivos por descomposición, mcd)

Sea la función **mcd** especificada en el apartado 3.1. Su demostración por inducción bien fundada más descomposición del primer parámetro provoca la creación de dos especificaciones inductivas parciales. La primera tiene asociado el cuadro parcial inicial recursivo:

asertos	objetivos	mcd (0,j)
if (u,v)<<(0,j) then if u>0 or v>0 then mcd(u,v)|u and mcd(u,v)|v and if w'|u and w'|v then w'≤mcd(u,v)		
0>0 or j>0		
	z|0 and z|j and if w|0 and w|j then w≤z	z

El segundo cuadro parcial es análogo, reemplazando **0** por **succ i.** ∎

USO DE INDUCCION ESTRUCTURAL

Hay ciertas especificaciones de función que se demuestran de una manera natural mediante inducción estructural. Una demostración por inducción estructural combina características de demostración por descomposición y por inducción bien fundada (ver apartado 2.6). Por tanto, el uso de la inducción estructural tiene como efecto en nuestro sistema deductivo la creación de varios cuadros parciales, algunos de ellos recursivos.

Definición (cuadro inicial recursivo estructural)

Dada, es una teoría, una especificación parcial inductiva estructuralmente:

$$(\forall\, \bar{x}{:}\bar{D})\ \left[\begin{array}{l}\text{if}\ \ \bar{Q}[\bar{y};\bar{f}\ \bar{y}]\\ \text{then}\ \ (\exists\, \bar{z}{:}\bar{R})\ Q[C\ s[\bar{x}];\bar{z}]\end{array}\right]$$

donde **C** es un constructor del tipo **D**,

$\bar{y}$ es el subconjunto de las variables de $\bar{x}$ que tienen tipo **D**,

$\bar{Q}[\bar{y};\bar{f}\ \bar{y}]$ es una conjunción de sentencias $Q[x_i;\bar{f}\ x_i]$ para

todas las variables x_i de tipo D,

un cuadro inicial recursivo estructural se define como:

asertos	objetivos	$\bar{f}$ C s[$\bar{a}$]
$\bar{Q}[\bar{b};\bar{f}\ \bar{b}]$		
E[C s[$\bar{a}$]]		
	S[C s[$\bar{a}$];$\bar{z}$]	$\bar{z}$

con tantos asertos $\bar{Q}[\bar{b};\bar{f}\ \bar{b}]$ como sentencias contenga $\bar{Q}$. ∎

Definición (derivación de funciones por inducción estructural)

Sea la especificación de unas funciones $\bar{f}$:

$$(\forall\ x_1{:}D_1,\ldots,x_n{:}D_n)\ (\exists\ \bar{z}{:}\bar{R})\quad Q[(x_1,\ldots,x_n);\bar{z}]$$

con una variable de entrada x_i de tipo D_i, o equivalentemente:

$$(\forall\ x_i{:}D_i)\ (\forall\ x_1{:}D_1,\ldots,x_n{:}D_n)\ (\exists\ \bar{z}{:}\bar{R})\quad Q[(x_1,\ldots,x_n);\bar{z}]$$

donde x_i no aparece dentro del segundo cuantificador universal. Supongamos que la teoría del tipo D_i contiene un principio de inducción estructural:

if S_1 and ... and S_n
then $(\forall\ x_i{:}D_i)\ F[x_i]$

donde cada conjuntado S_j representa bien un caso básico, es decir, una sentencia

$$(\forall\ \bar{x}{:}\bar{D})\ F[s[\bar{x}]]$$

bien un caso inductivo, es decir, una sentencia

$$(\forall\ \bar{x}{:}\bar{D})\ \left[\begin{array}{l}\text{if } \bar{F}[\bar{y}]\\ \text{then } F[C\ s[\bar{x}]]\end{array}\right]$$

donde **C** es un constructor del tipo D_i,

$\bar{x}$ es un conjunto de variables $x_1,\ldots,x_l$

$\bar{y}$ es el subconjunto de las variables de $\bar{x}$ que tienen tipo D_i, y

$\bar{F}[\bar{y}]$ es una conjunción de sentencias $F[x_k]$ $(1\leq k\leq l)$ para todas las variables de $\bar{x}$ de tipo D_i.

Las funciones $\bar{f}$ especificadas se derivan construyendo **n** cuadros parciales. Por cada caso básico del principio de inducción estructural se crea una cuadro parcial inicial:

asertos	objetivos	$\bar{f}(a_1,\ldots,s[\bar{a}],\ldots,a_n)$
	$Q[(a_1,\ldots,s[\bar{a}],\ldots,a_n);\bar{z}]$	$\bar{z}$

Asimismo, por cada cuadro inductivo del principio de inducción estructural se crea una cuadro parcial inicial recursivo estructural:

asertos	objetivos	$f\ (a_1,\ldots,C\ s[\bar{a}],\ldots,a_n)$
$\bar{Q}[(a_1,\ldots,\bar{b},\ldots,a_n);\bar{f}\ (a_1,\ldots,\bar{b},\ldots,a_n)]$		
	$Q[(a_1,\ldots,C\ s[\bar{a}],\ldots,a_n);\bar{z}]$	$\bar{z}$

donde se incluyen tantos asertos inductivos como variables pertenecientes a $\bar{y}$ hay. ∎

Proposición (corrección de programa derivado estructuralmente)

En cualquier teoría,

si los términos básicos $\bar{t}_j[\bar{a}_j]$ satisfacen el cuadro

parcial inicial recursivo estructural de cabecera $\bar{f}\ s_j[\bar{a}_j]$ $(1 \leq j \leq n_i)$

y la teoría del tipo D_i contiene un principio de inducción estructural con **n** casos $(1 \leq i \leq m)$,

entonces el programa de igualdades $\bar{f}\ s_j[\bar{a}_j] = \bar{t}_j[\bar{a}_j]$ satisface la especificación.

Demostración:

La demostración es similar a la de la proposición de corrección de programa derivado por descomposición. Se diferencia en la invocación del principio de inducción estructural (en vez del teorema de descomposición). Como consecuencia algunos de los conjuntados tienen el formato inductivo:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \left[\begin{array}{l}\text{if } \bar{Q}[\bar{y};\bar{f}\ \bar{y}] \\ \text{then } Q[s_j[\bar{x}_j];\bar{f}\ s_j[\bar{x}_j]]\end{array}\right]$$

en vez del formato descompositivo:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ Q[s_j[\bar{x}_j];\bar{f}\ s_j[\bar{x}_j]]$$

Los conjuntados no inductivos se tratan igual que en el caso descompositor. Si el caso inductivo **j** tiene unos valores de función definidos por:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \bar{f}\ s_j[\bar{x}_j] = \bar{t}_j[\bar{x}_j]$$

puede formularse equivalentemente como:

$$(\forall\ \bar{x}_j{:}\bar{D}_j)\ \left[\begin{array}{l}\text{if } \bar{Q}[\bar{y};\bar{f}\ \bar{y}] \\ \text{then } Q[s_j[\bar{x}_j];\bar{t}_j[\bar{x}_j]]\end{array}\right]$$

que (por la propiedad del término de salida parcial), es cierto si los términos $\bar{t}_j[\bar{x}_j]$ satisfacen el cuadro parcial recursivo estructural de cabecera $\bar{f}\ s_j[\bar{a}_j]$, como hemos supuesto.

Si el caso inductivo j no tiene ningún valor definido, podemos asociar a las funciones unos valores constantes $\bar{c}_j$ cualesquiera y expresar el caso:

$$(\forall\, \bar{x}_j{:}\bar{D}_j)\ \left[\begin{array}{l}\text{if } \bar{Q}[\bar{y};\bar{f}\ \bar{y}]]\\ \text{then } Q[s_j[\bar{x}_j];\bar{c}_j]\end{array}\right]$$

que (de nuevo por la propiedad del término de salida parcial) es cierto si los términos $\bar{c}_j$ satisfacen el cuadro inicial recursivo de la especificación parcial correspondiente. Pero como es una especificación parcial obviamente válida, cualesquiera términos básicos $\bar{c}_j$ satisfacen su cuadro inicial. ∎

Ejemplo (cuadros parciales por inducción estructural, ordinsercion)

Sea la función de ordenación de una lista de enteros, que la llamamos **ordinsercion** si utiliza una estrategia de ordenación por inserción. Su especificación no endulzada es:

(∀ l:lista) (∃ z:lista) perm(l,z) and ord(z)

Según el principio de inducción estructural sobre listas con un caso básico, para demostrar una sentencia cuantificada sobre una lista, basta demostrar dos casos, uno inductivo y otro no. En consecuencia deben demostrarse dos especificaciones parciales:

(∃ z:lista) perm(nil,z) and ord(z)

y

(∀ x:entero;l:lista) (∃ z:lista)
[if perm (l,ordinsercion l) and ord ordinsercion l
[then perm (x::l,z) and ord z]

Sus cuadros parciales respectivos son:

asertos	objetivos	ordinsercion nil
	perm(nil,z) and ord z	z

y

asertos	objetivos	ordinsercion x::l
perm(l,ordinsercion l)		
ord ordinsercion l		
	perm(x::l,z) and ord z	z

3.7. DERIVACION DE FUNCIONES CON RANGO TUPLA

Los mecanismos para derivar ecuaciones con patrones en sus cabeceras se han desarrollado en los apartados anteriores. Sin embargo, no hemos contemplado aún la obtención de patrones en definiciones locales. Este problema suele ir ligado a la derivación de funciones con rango tupla.

ESPECIFICACION DE FUNCION CON RANGO TUPLA

Una especificación de función declara el comportamiento de ésta mediante el conocido formato:

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ \bar{z}:\bar{R})\ \left[\begin{array}{l}\text{if } E[\bar{x}]\\ \text{then } S[\bar{x};\bar{z}]\end{array}\right]$$

es decir:

$$(\forall\ x_1:D_1,\ldots,x_n:D_n)\ (\exists\ \bar{z}:\bar{R})\ \left[\begin{array}{l}\text{if } E[(x_1,\ldots,x_n)]\\ \text{then } S[(x_1,\ldots,x_n);\bar{z}]\end{array}\right]$$

Puede observarse que nos referimos a la salida mediante unas variables $\bar{z}$ que deben satisfacer la relación de entrada-salida **S**. Supongamos por sencillez (y sin pérdida de generalidad) que la especificación contiene una sola variable **z** de salida. Es corriente que la salida **z** de la función sea de un tipo estructurado y la relación **S** no se establezca realmente entre los parámetros de entrada y el valor global de **z**, sino entre aquéllos y los componentes de **z**. Un caso usual es cuando el rango de la función es una tupla, caso al que dedicamos este apartado.

Ejemplo (especificación seleccionar)

Sea una función **menoresymayores** que, dado un entero **x** y una lista **l** de enteros, devuelve un par de listas que respectivamente contienen los elementos de **l** que son menores y mayores que **x**. Una especificación endulzada es:

```
menoresymayores (x,l):entero#lista
   <= encontrar z:lista#lista
      tal que perm (l,primero z<>segundo z) and
              (∀ w:entero) [ if w∈primero z then w≤x and ]
                           [ if w∈segundo z then x≤w     ]
```

Si la función no utiliza ninguna función auxiliar que devuelva tuplas, es obvio que debe construir en su cuerpo la tupla de salida. La parte de la demostración de la especificación orientada a la formación de la tupla de salida es rutinaria, obedeciendo siempre al mismo esquema. Podemos ganar claridad de la especificación y ahorrar pasos de deducción si endulzamos la especificación para las funciones con rango tupla. Sin embargo, el nuevo formato de especificación, a diferencia del usado para funciones con dominio tupla, no es de aplicación general, sino sólo cuando se cumplen las restricciones mencionadas.

Definición (especificación con rango tupla)

Una especificación de función con rango de tipo tupla y con referencias a la variable **z** de salida por medio de funciones selectoras puede tener el formato:

$$f\ \bar{x}:\bar{D} \Leftarrow \text{encontrar } (z_1,\ldots,z_n):R_1\#\ldots\#R_n \text{ tal que } S[\bar{x};z_1,\ldots,z_n] \text{ donde } E[\bar{x}]$$

y por tanto el cuadro inicial tiene como objetivo:

	$S[\bar{x};z_1,\ldots,z_n]$	$(z_1,\ldots,z_n)$

Justificación:

Sea una especificación:

$$(\forall\ \bar{x}:\bar{D})\ (\exists\ z:R_1\#\ldots\#R_n)\ \text{if } E[\bar{x}] \text{ then } S[\bar{x};z]$$

El cuadro inicial contiene como objetivo:

	$S[\bar{x};z]$	z

o identificando las apariciones de **z** dentro de una aplicación de función selectora de tuplas y en cualquier otro contexto,

	$S[\bar{x};z,$primero z,...,n-ésimo z$]$	z

Aplicando reemplazamiento por igualdad entre este objetivo y el axioma **primero**, obtenemos el objetivo:

	$S[\bar{x};(z_1,\ldots,z_n),z_1,\ldots,$n-ésimo$(z_1,\ldots,z_n)]$	$(z_1,\ldots,z_n)$

Aplicando reiteradas veces reemplazamiento por igualdad con los axiomas de las demás funciones selectoras obtenemos:

	$S[\bar{x};(z_1,\ldots,z_n),z_1,\ldots,z_n)]$	$(z_1,\ldots,z_n)$

donde si todas las consultas de la variable de salida **z** son por medio de funciones selectoras, el objetivo tiene el formato:

	$S[\bar{x};z_1,\ldots,z_n)]$	$(z_1,\ldots,z_n)$

Este es el cuadro correspondiente a la nueva especificación. Puede verse que su uso permite ahorrar varios pasos de deducción. Asimismo la especificación asociada es más sencilla que la especificación sin endulzar. ∎

Los nuevos formatos de especificación y cuadro inicial pueden usarse tanto para derivaciones completas como parciales. Debe recordarse que una especificación como la anterior es un nuevo endulzamiento de la especificación verdadera. Por tanto, cualquier sentencia que haga referencia a la especificación de la función (p.ej. hipótesis de inducción o la misma sentencia de corrección de la función una vez derivada ésta) debe conservar su formulación inicial con funciones selectoras.

La generalización del formato a la derivación simultánea de varias funciones es fácil:

$f_1,\ldots,f_n\ \bar{x}:\bar{D}$ <= encontrar $p_1:R_1;\ldots;p_n:R_n$
tal que $S[\bar{x};vars(p_1,\ldots,p_n)]$
donde $E[\bar{x}]$

donde p_i $(1\leq i\leq n)$ representa $z_i:R_i$ si la i-ésima variable de salida no tiene tipo tupla o

$(z_{i1},\ldots,z_{ini}):R_{i1}\#\ldots\#R_{ini}$ en caso afirmativo.

vars$(p_1,\ldots,p_n)$ representa la secuencia de todas las variables z_{ij} $(1\leq i\leq n,\ 1\leq j\leq n_i)$ que aparecen en la declaración del tipo rango de las funciones.

Ejemplo (especificación seleccionar, endulzada)

La especificación de la función **seleccionar** puede expresarse según el nuevo formato, resultando:

```
menoresymayores (x,l):entero#lista
   <= encontrar (z1,z2):lista#lista
      tal que  perm(l,z1<>z2) and
               (∀ w:entero) [ if wεz1 then w≤w  and ]
                            [ if wεz2 then x≤w      ]
```

Asimismo, si la demostración de la especificación la hacemos mediante inducción estructural sobre listas, el cuadro inicial del caso recursivo es:

asertos	objetivos	me... (x,y::l)	
perm(l,primero me... (x,l)<>segundo me... (x,l))			
if uεprimero menoresymayores (x,l) then u≤x			
if uεsegundo menoresymayores (x,l) then x≤u			
	perm(l,z_1<>z_2) and if wεz_1 then w≤x and if wεz_2 then x≤w	(z_1,z_2)	

Obsérvese el distinto aspecto presentado por las subsentencias correspondientes de la hipótesis de inducción y el objetivo inicial. La hipótesis inicial se ha dividido automáticamente en tres asertos por la regla de partición-and. ∎

El endulzamiento de especificación descrito puede aplicarse de forma similar a valores de salida con otros tipos

estructurados (p.ej. listas) de los que interesan sus componentes. Sin embargo, dichas situaciones son más raras, por lo que no las tratamos.

DERIVACION DE DEFINICIONES LOCALES CON PATRONES

Hemos visto que la especificación de funciones con rango tupla (sin el endulzamiento anterior) suele incluir funciones selectoras de tuplas. Como consecuencia, el uso durante una derivación de dichas sentencias de especificación (en hipótesis de inducción o sentencias de corrección) puede provocar la derivación de términos de salida con funciones selectoras.

Ejemplo (derivación menoresymayores, recursión)

Sea una función **menoresymayores** antes especificada. Su demostración por inducción estructural sobre su parámetro de tipo lista crea una cuadro parcial recursivo inicial que eventualmente podemos hacer que contenga el objetivo final:

	true	if y≤x then (y::primero me... (x,1), segundo me... (x,1)) else (primero me... (x,1), y::segundo me... (x,1))

∎

Aunque correcto, no corresponde a un estilo de programación funcional moderna el uso de funciones selectoras, sino que se evitan utilizando patrones. El uso de varias ecuaciones con patrones evita la aplicación de dichas funciones a los parámetros de las funciones. En caso de que las funciones selectoras aparezcan en subtérminos del lado derecho de una ecuación, se evita usando definiciones locales con patrones.

Ampliamos el sistema deductivo para introducir definicio-

nes locales con patrones en los términos de salida añadiendo la reescritura:

$$t[sel_1\ t',\ldots,sel_n\ t'] \ \Leftrightarrow\ \text{let } C(x_1,\ldots,x_n) == t' \text{ in } t[x_1,\ldots,x_n]$$

donde **t'** es de tipo **T**, $x_1,\ldots,x_n$ son de tipos respectivos $T_1,\ldots,T_n$, **C** es un constructor de tipo $T_1\#\ldots\#T_n \rightarrow T$ y $sel_1,\ldots,sel_n$ son funciones selectoras de tipo respectivo $T\rightarrow T_1,\ldots,T\rightarrow T_n$.

En el caso particular de las tuplas tenemos la reescritura:

$$t[\text{primero } t',\ldots,\text{n-ésimo } t'] \ \Leftrightarrow\ \text{let } (x_1,\ldots,x_n) == t' \text{ in } t[x_1,\ldots,x_n]$$

Podíamos haber declarado la regla como una simplificación, en vez de reescritura. El proceso simplificador termina eventualmente porque cada simplificación reduce el número de funciones selectoras. Esta decisión tiene como ventaja que la regla se aplica automáticamente. Sin embargo presenta la desventaja de que puede aplicarse prematuramente, es decir, que posteriormente el término de salida puede contener más apariciones de funciones selectoras. Esto obliga a formar varias definiciones locales, resultando una ecuación poco clara. Una solución es disponer de simplificaciones que manipulen definiciones locales, pero su diseño no es evidente. En conclusión, parece preferible que sea el programador quien decida cuándo se introducen las definiciones locales.

Ejemplo (derivación menoresymayores, recursión con definiciones locales)

Aplicando la reescritura descrita al objetivo final previo, obtenemos como nuevo objetivo:

	true	let (p,s) == menoresymayores (x,l) in if y≤x then (y::p,s) else (p,y::s)

3.8. DERIVACION DE VARIABLES ANONIMAS Y SINONIMAS

Desarrollamos en este apartado la última característica relacionada con los patrones, ésta propia de Hope (y por consiguiente de miniHope): el uso de variables anónimas y sinónimas.

La clase de variables en cuestión se puede obtener modificando directamente las ecuaciones derivadas. Veamos informalmente el proceso con un ejemplo, y después formulamos el mecanismo concreto de derivación.

Ejemplo (ordmezcla)

Sea una función de ordenación de listas por mezcla, que consta de las tres ecuaciones siguientes:

```
--- ordmezcla nil
      <= nil ;
--- ordmezcla x::nil
      <= x::nil ;
--- ordmezcla x::y::l
      <= mezclar(ordmezcla(izdo(x::y::l)),
                 ordmezcla(dcho(x::y::l))) ;
```

El patrón de las ecuaciones segunda y tercera contienen subtérminos no variables (**x::nil** y **x::y::l**, respectivamente) que se utilizan en la parte derecha de sus ecuaciones. Por tanto, podemos declarar una variable sinónima **s** en cada una de las ecuaciones y utilizarla consistentemente en su lado derecho. El programa queda:

```
--- ordmezcla nil
      <= nil ;
--- ordmezcla s & x::nil
      <= s ;
--- ordmezcla s & x::y::l
      <= mezclar(ordmezcla(izdo(s)),
                 ordmezcla(dcho(s))) ;
```

Ahora resulta que las dos ecuaciones modificadas contienen variables en sus cabeceras que no son citadas en sus cuerpos. Podemos hacer anónimas estas variables, resultando el programa:

```
--- ordmezcla nil
      <= nil ;
--- ordmezcla s & _::nil
      <= s ;
--- ordmezcla s & _::_::_
      <= mezclar(ordmezcla(izdo(s)),
                 ordmezcla(dcho(s))) ;
```

El patrón de la segunda ecuación indica que el parámetro es una lista atómica, pero en el cuerpo no importa quién es su único elemento, sino la lista en conjunto. Lo mismo ocurre en la tercera ecuación, sólo que en este caso el parámetro lista debe contener al menos dos elementos.

Debe observarse que se han podido introducir variables anónimas gracias a la introducción previa de variables sinónimas. ■

El ejemplo ilustra que la introducción de ambas construcciones sintácticas debe hacerse en cierto orden (primero variables sinónimas, segundo variables anónimas) para garantizar la introducción de todas las variables anónimas posibles. Aunque la ecuación va a sufrir una sucesión de modificaciones hasta obtener su aspecto final, no escribimos las ecuaciones intermedias.

DERIVACION DE VARIABLES SINONIMAS

Hay razones de eficiencia y legibilidad para introducir variables sinónimas. Sin embargo, no siempre interesa introducir todas las variables sinónimas posibles porque conduce a una ecuación que tiene un patrón poco legible y con algunas variables sinónimas poco relevantes. Por tanto es difícil establecer un criterio puramente sintáctico que determine las variables sinónimas por introducir en la ecuación. Es preferible que el programador haga la elección, un tanto subjetiva, de identificar los subtérminos de la cabecera que van a tener sinónimo; estos subtérminos deben aparecer en el término de salida del objetivo final. La construcción de la ecuación con variables sinónimas es automática.

Definición (formación de ecuación con variables sinónimas)

Supongamos que tenemos un cuadro final (parcial o completo), correspondiente a la derivación de una ecuación de una función **f** con patrón t'_1 y términos de salida t'_2. Supongamos también que se quiere introducir en la ecuación una variable **s** sinónima de un subtérmino no variable t'_3 del patrón de la ecuación.

El procedimiento para crear la ecuación correspondiente al cuadro con la variable sinónima **s** es:

(i) Se crea la ecuación de la manera usual, obteniendo:

$$--- \; f \; t_1 <= t_2 \; ;$$

donde t_1 y t_2 se obtienen respectivamente de t'_1 y t'_2 sustituyendo las constantes de Skolem (los parámetros) por variables. También podemos expresar la ecuación como:

$$--- \; f \; t_1[t_3] <= t_2[t_3]] \; ;$$

donde t_3 se ha obtenido a partir de t'_3 sustituyendo las constantes por variables, y puesto que t'_3 es un subtérmino de t'_1 y t'_2.

(ii) Se introduce la variable sinónima en el lado izquierdo de la ecuación y se reemplaza cada aparición en el lado derecho del subtérmino en cuestión por la variable sinónima. Es decir, la ecuación anterior se transforma en:

$$\text{--- f } t_1[s \ \& \ t_3] <= t_2[s] \ ;$$

Estos dos pasos se efectúan secuencial y automáticamente, así que sólo se muestra la ecuación final, sin hacer referencia a la ecuación intermedia del paso (i).

La introducción de varias variables sinónimas se hace igual, sólo que la introducción y reemplazamiento es simultáneo para todas ellas. Si tenemos varias funciones f, el proceso se realiza independientemente para la ecuación de cada función. ∎

Ejemplo (ordmezcla, derivación de variables sinónimas)

La función **ordmezcla** antes expuesta puede derivarse mediante inducción bien fundada más descomposición en tres casos. La demostración de la especificación parcial correspondiente a una lista de más de un elemento puede conducir al objetivo final:

asertos	objetivos	ordmezcla a::b::t
	. . .	
	true	mezclar (ordmezcla izdo a::b::t, ordmezcla dcho a::b::t)

Supongamos que queremos introducir la variable **s** sinónima del subtérmino **a::b::t**. El procedimiento antes descrito permite extraer del cuadro parcial la ecuación:

```
--- ordmezcla s & x::y::l
      <= mezclar (ordmezcla izdo s,ordmezcla dcho s) ;
```

DERIVACION DE VARIABLES ANONIMAS

La derivación de variables anónimas supone la adición de un paso al procedimiento previo.

Definición (formación de una ecuación con variables sinónimas y anónimas)

Sea un cuadro final, correspondiente a la derivación de la ecuación de una función **f** con patrón t'_1 y término de salida t'_2 y unas variables $\bar{s}$ sinónimas de unos subtérminos $\bar{t}'_3$.

El procedimiento para crear la ecuación correspondiente al cuadro con las variables sinónimas $\bar{s}$ y variables anónimas es:

(i) y (ii) Igual que antes.

(iii) Supongamos que el lado izquierdo de la ecuación es **f** t_1 y el lado derecho, t_2. Realizamos dos subpasos:
  (a) Se determina el conjunto **C** de variables no sinónimas de la cabecera **f** t_1.
  (b) Determinamos si cada variable $v \epsilon C$ aparece en el lado derecho t_2 de la ecuación. Si no aparece reemplazamos el patrón $t_1[v]$ en la ecuación por $t_1[\_]$.

En el subpaso (iii.a) no se consideran las variable sinó-

nimas puesto que, por construcción, éstas siempre aparecen en el lado derecho de la ecuación. Los tres pasos se efectúan secuencial y automáticamente, así que sólo se mostrará la ecuación final, sin hacer referencia a las ecuaciones intermedias de los pasos (i) y (iii).

Si tenemos varias funciones $\bar{f}$, el proceso se realiza independientemente para la ecuación de cada función. ∎

Ejemplo (ordmezcla, derivación de variables sinónimas y anónimas)

Dada la ecuación:

```
--- ordmezcla s & x::y::l
        <= mezclar (ordmezcla izdo s,ordmezcla dcho s) ;
```

el conjunto de variables no sinónimas de la cabecera es {x,y,l}. Ninguna de las variables de este conjunto aparece en el lado derecho de la ecuación, por lo que son sustituidas en la cabecera por el carácter de subrayado. La ecuación definitiva es:

```
--- ordmezcla s & _::_::_
        <= mezclar (ordmezcla izdo s,ordmezcla dcho s) ;
```

∎

3.9. SUMARIO DEL PROCESO DE DERIVACION

Tras examinar las diversas facetas del proceso de derivación de funciones conviene hacer una recapitulación del proceso completo. A continuación describimos el procedimiento de derivación examinando con cierto detalle dos actividades corrientes, la introducción de llamadas recursivas y de funciones auxiliares. Este procedimiento se utiliza en el próximo

capítulo para la derivación de varios programas.

PROCESO DE DERIVACION

Veamos los pasos a seguir siguiendo su orden de realización.

Teoría lógica

La sentencia de especificación y su demostración deben hacerse dentro de una teoría lógica. Dado que miniHope incluye varios tipos predefinidos (enteros no negativos, listas de enteros y tuplas de elementos de estos tipos), podemos suponer que su teoría combinada (consultar Apéndice B) siempre está disponible.

El programador también puede derivar funciones con tipos nuevos. El nuevo tipo debe declararse según las reglas sintácticas de miniHope y posteriormente debe desarrollarse su teoría lógica correspondiente. Si el tipo es un álgebra de palabras, la teoría se desarrolla automáticamente siguiendo las directrices dadas en el apartado 2.4. Si se desea que el tipo corresponda a una estructura algebraica distinta, no hay normas fijas, por lo que la formulación de la teoría puede ser una tarea difícil. Sin embargo se conservan las directrices referentes a los tipos de los constructores y los axiomas de la igualdad. Cuestiones corrientes a plantearse durante la construcción de la teoría son la necesidad de un principio de inducción estructural, la relación entre los distintos constructores o la existencia de valores prohibidos en el tipo. En cualquier caso, además de los axiomas de la teoría es corriente incluir teoremas adicionales, como teoremas de descomposición o propiedades de las funciones. La teoría desarrollada se añade a la teoría combinada de los tipos existentes previamente.

Especificación de funciones

Las funciones que queremos derivar se especifican con el formato ampliado para la especificación de funciones con dominio o rango de tipo tupla:

$f_1,\ldots,f_n$ $x_1{:}D_1,\ldots,x_m{:}D_m$
<= encontrar
$(z_{11},\ldots,z_{11l}){:}R_{11}\#\ldots\#R_{11l},\ldots,(z_{n1},\ldots,z_{nln}){:}R_{n1}\#\ldots\#R_{nln}$
tal que
$S[(x_1,\ldots,x_m);z_{11},\ldots,z_{11l},\ldots,z_{n1},\ldots,z_{nln}]$
donde
$E[(x_1,\ldots,x_m)]$

Este esquema de especificación declara el comportamiento de **n** funciones $f_1,\ldots,f_n$, todas con dominio de tipo $D_1\#\ldots\#D_m$. El rango de cada función f_i ($1\leq i\leq n$) es de tipo $R_{i1}\#\ldots\#R_{ili}$. Este formato es correcto para funciones con dominio tupla (ver apartado 2.2), pero para funciones con rango tupla sólo es conveniente si los términos de salida se van a formar con el constructor coma de tuplas (ver apartado 3.7).

Un mismo programa puede tener distintas especificaciones, (véase un ejemplo en [MaWa87]), sin que pueda decirse a priori cúal es la mejor. Asimismo es difícil desarrollar una especificación correcta y completa. Teniendo en cuenta que la especificación es la base de todo el proceso deductivo (en general de cualquier proceso transformativo), se comprende la importancia de esta tarea.

La especificación contiene una información que se expresa automáticamente como parte del programa final: la declaración de tipo de las funciones. Es decir, se genera una declaración con el esquema:

dec f_i : $D_1\#\ldots\#D_m$ -> $R_{i1}\#\ldots\#R_{ini}$;

para cada función f_i.

También debe determinarse el conjunto de funciones que pueden aparecer en el cuerpo del programa derivado, es decir, el conjunto de funciones primitivas en esta derivación. En general se supone que todas las funciones definidas en la teoría son primitivas, salvo que se diga lo contrario. En todo caso, siempre se consideran automáticamente como primitivas las funciones a derivar en un cuadro inicial recursivo. También se consideran automáticamente como primitivas las funciones auxiliares derivadas.

Formación del cuadro inicial

La sentencia de especificación puede demostrarse de varias maneras que surgen de tomar dos decisiones sobre la demostración: si será inductiva y si será descompositiva. La primera cuestión implica descartar la inducción o elegir un esquema de inducción (bien fundada o estructural sobre ciertos parámetros y con cierto número de casos básicos). La segunda cuestión implica decidir los parámetros a descomponer en distintos constructores según sea su tipo. Ambas cuestiones no son excluyentes (salvo en caso de inducción estructural, que ya incorpora implícitamente una descomposición), así que deben decidirse en el orden mencionado. La decisión del esquema de demostración no es fácil. A veces puede guiar la "intuición" del esquema del programa final. Puesto que la demostración es un proceso tentativo, es normal que deba abandonarse un esquema de demostración para adoptar otro diferente.

Según el esquema de deducción adoptado, el número y formato de los cuadros iniciales tienen características cambiantes. Una demostración por inducción bien fundada (véase 2.5 y 3.6) supone la inclusión de una hipótesis de inducción. Una demostración por inducción estructural (véase 2.5 y 3.6) supone la demostración de varias especificaciones parciales, algunas con hipótesis de inducción. La inclusión de una hipótesis de inducción implica añadir uno o varios asertos. Una demostración por descomposición (véase 2.6 y 3.5) supone la

demostración de varias especificaciones parciales, una por cada constructor utilizado en la formación de los parámetros elegidos. La demostración de varias especificaciones parciales implica construir varios cuadros iniciales, uno por especificación. Cada cuadro parcial se parte y simplifica lo máximo posible; también se eliminan los asertos que son instancias de sentencias válidas de la teoría.

En general el formato de un cuadro inicial obedece a las siguientes reglas (véanse los apartados 3.3, 3.5, 3.6 y 3.7):

- Las columnas de salida contienen la cabecera de la ecuación a derivar, es decir, el símbolo de función y el patrón del parámetro.
- Existe un objetivo único que contiene la relación de entrada-salida. El término de salida en la columna de la función f_i es z_i si su rango no es de tipo tupla y $(z_{i1},\ldots,z_{ili})$ si es de tipo tupla (cuando interese, como se explica en el apartado 3.7).
- Pueden existir varios asertos, sin término de salida. Estos asertos expresan hipótesis de inducción o condiciones de entrada sobre los parámetros. Los axiomas y teoremas de la teoría se supone que implícitamente aparecen como asertos. En las hipótesis de inducción la relación de entrada-salida se expresa en términos de la aplicación de función, no de un hipotético valor de salida, es decir, tienen el formato $S[\bar{x};\hat{f}\ \bar{x}]$ en vez de $S[\bar{x};\bar{z}]$.
- Las variables del patrón de la ecuación aparecen como constantes de Skolem y las variables de salida como variables libres.

Derivación del programa

Se demuestra la validez de cada cuadro. Es difícil dar reglas sobre cómo demostrar su validez. Si tenemos en mente un esquema del programa final, puede proporcionarnos cierta

orientación. El esquema nos llevará a buscar una demostración en la que intervengan las hipótesis de inducción, a varias ramas de demostración con objetivos "terminales" excluyentes, a reformular un objetivo como otra sentencia equivalente pero ejecutable, etc. Algunos de estos pasos son similares a las estrategias del método de Bibel [Bibel80].

Más fácil es identificar sintácticamente situaciones donde puede ser conveniente introducir llamadas recursivas o derivar funciones auxiliares. Por su importancia las describimos por separado al final del apartado; por sencillez no se considera el uso de patrones, aunque pueden incorporarse sin problemas. Estas heurísticas fueron desarrolladas por Manna y Waldinger al construir un sistema transformativo no puramente deductivo [MaWa79]; se han descrito con bastante detalle en otros sitios [MaWa80, MaWa87, Traugott89].

Durante el proceso de derivación el programador puede decidir la introducción, mediante reescritura, de definiciones locales. Aunque puede hacerse en cualquier momento de la derivación, es más seguro realizarlo una vez obtenido el cuadro final. Esta reescritura puede provocar la eliminación de funciones selectoras del término de salida.

Una vez deducido un cuadro final, se forma la ecuación correspondiente. La ecuación se construye de manera automática a partir del cuadro final y la identificación de subtérminos de la cabecera por redenominar con variables sinónimas. El proceso de creación de la ecuación se expone en el apartado 3.8, consistiendo básicamente en el reemplazando las constantes de Skolem por variables, la adición de variables sinónimas y anónimas en la cabecera de la ecuación y el reemplazamiento en el cuerpo de la ecuación de ciertos subtérminos por las variables sinónimas equivalentes.

Ampliación de la teoría

La teoría utilizada para la derivación de las funciones se amplía con éstas, añadiendo dos tipos de axiomas (véanse los apartados 2.3 y 2.4):

- Axiomas de definición del programa. Se introduce un axioma por ecuación. Cada axioma es una igualdad entre las correspondencias lógicas de la cabecera y el cuerpo de la ecuación, es decir,
  $$(\forall\ \bar{x}:\bar{D})\ f\ s[\bar{x}] = t[\bar{x}]$$
  donde **s** es un término formado por constructores y las variables $\bar{x}$, y **t** es un término cualquiera.
- Teoremas de corrección del programa. Se introduce un axioma por función que expresa que la función satisface su especificación, es decir,
  $$(\forall\ \bar{x}:\bar{D})\ Q[\bar{x};f\ \bar{x}]$$

INTRODUCCION DE LLAMADAS RECURSIVAS

Veamos una situación frecuente de introducción de llamadas recursivas. Por generalidad nos centramos en la inducción bien fundada; en la inducción estructural el aserto es similar, sólo que el antecedente $\bar{x}<<\bar{a}$ no existe, ya que como relación << se ha tomado la relación "estructural" y los términos $\bar{x}$ y $\bar{a}$ elegidos satisfacen obviamente la relación. También consideramos una sola variable de entrada, una sola variable de salida y una sola función, sin pérdida de generalidad.

Sea un cuadro inicial que incluye como asertos la hipótesis de inducción y la condición de entrada y como objetivo la condición de entrada-salida:

if x<<a then if E[x] then S[x,f x]		
E[a]		
	S[a,z]	z

Supóngase que durante la demostración se obtiene un objetivo que contiene una instancia de la condición de entrada-salida:

	O[S[s,z']]	t[z']

donde **s** y **t** son términos y **z'** es una variable.

Podemos aplicar la regla de resolución entre el objetivo y la hipótesis de inducción con un unificador más general θ = {x<-s, z'<-f s}, obteniendo (tras simplificaciones true-false):

	O[true] and s<<a and E[s]	t[f s]

La subsentencia **O[true]** del nuevo objetivo indica que la subsentencia **S[s,z']** del objetivo **O[S[s,z']]** ha sido satisfecha por la hipótesis de inducción. La subsentencia **E[s]** del objetivo asegura que la función se aplica recursivamente a un término que satisface la condición de entrada. La condición **s<<a** asegura que la llamada recursiva se aplica a un dato "menor" según <<, y por tanto la función alguna vez termina su ejecución. La relación bien fundada << puede ser cualquiera y no necesita ser elegida hasta más adelante en la prueba.

La heurística puede generalizarse permitiendo que la hipótesis de inducción se use cuando una subsentencia del objetivo

actual sea una instancia, no del objetivo inicial, sino de una subsentencia del objetivo inicial.

INTRODUCCION DE FUNCIONES AUXILIARES

Es corriente que, en un programa funcional, unas funciones utilicen en su cuerpo a otras funciones no triviales. Durante la derivación deductiva de las primeras funciones, pueden introducirse las otras funciones en sus términos de salida mediante resolución de parte del objetivo con sus respectivos teoremas de corrección. Si estas funciones no estuvieran desarrolladas, deben a su vez derivarse deductivamente; se dice que las nuevas funciones son auxiliares de las funciones principales. La derivación de una función auxiliar obliga a que el proceso deductivo en marcha se interrumpa, realizándose el correspondiente a la función auxiliar y, una vez acabado éste, continuando la deducción principal donde se dejó. (Obsérvese que normalmente sólo tiene sentido derivar una función auxiliar si es recursiva, ya que en caso contrario es un término que puede introducirse por instanciación de variables en la función principal.)

Supongamos que durante la derivación de unas funciones **f** se obtienen las filas:

E[t[a]]		
	O[S[t[a],z]]	r[z]

donde **t[a]** es un término básico y **z** es una variable.

Supongamos que se dispone del teorema de corrección de una función **f'**, considerada primitiva en esta derivación:

if E[x] then S[x,f' x]		

Mediante sucesivas resoluciones del objetivo con el aserto **E[t[a]]** y el teorema de corrección obtenemos el nuevo objetivo:

	O[true]	r[f' t[a]]

que puede conducir a un objetivo final:

	true	r'[f' t[a]]

obteniendo el programa

```
--- f x <= r'[f' t[x]] ;
```

En caso de que la función auxiliar **f'** no esté desarrollada, debe derivarse mediante un cuadro deductivo auxiliar. Su sentencia de especificación es:

```
(∀ x:D) (∃ z:R)  if E[x] then S[x;z]
```

Una función disponible puede introducirse según el mecanismo descrito siempre que se quiera. Sin embargo, hay una táctica para detectar situaciones en la que conviene derivar una función auxiliar no disponible. Informalmente se basa en la posibilidad de razonar inductivamente, no sobre la hipótesis de inducción, sino sobre un objetivo previo. En términos de programas esto significa que se recurra sobre un trozo de código distinto de la función en curso de derivación, que obviamente debe ser una función auxiliar.

Supongamos que a partir del subcuadro formado por las filas (antes citadas):

E[t[a]]		
	O[S[t[a],z]]	r[z]

obtenemos una fila con un objetivo de la forma:

	O'[S[t'[a],z']]	r'[z']

donde **t'[a]** es un término básico y **z'** es una variable. El nuevo objetivo contiene una réplica de una parte del objetivo previo; ambas réplicas difieren en los términos **t[a]** y **t'[a]** y las variables **z** y **z'**.

Esto sugiere la introducción de una función auxiliar nueva **f'**, cuyo teorema de especificación es:

if E[x] then S[x,f' x]		

Para derivar esta función construimos el cuadro auxiliar inicial:

if y<<b then if E[y] then S[y,f' y]		
E[b]		
	S[b,z]	z

Si podemos repetir la derivación realizada en el cuadro original, se obtiene un objetivo **O'[S[t''[b],z']]**:

	O''[S[t''[b],z']]	r''[z']

que podemos satisfacer mediante la hipótesis de inducción, aplicando resolución entre ambas filas con unificador θ = {y<-t''[b], z'<-f' t''[b]}. Obtenemos el nuevo objetivo:

	O'[true] and t''[b]<<b and E[t''[b]]	r''[f' t''[b]]

que previsiblemente conducirá a un objetivo final:

	true	s[f' t''[b]]

El programa derivado finalmente es:

```
--- f x <= r'[f' t[x]] ;
--- f' x <= s[f' t''[x]] ;
```

Conviene hacer dos puntualizaciones sobre esta táctica. Primero, puede ocurrir que se precise la inclusión en el cuadro auxiliar de un aserto **E'[b]**, tal que **E'[t[a]]** estaba presente en el cuadro original. Es decir, añadimos **E'[b]** a la condición de entrada de la especificación de **f'**. Esto implica modificar ciertas partes de la prueba para mantener la consistencia, añadiendo la nueva condición al aserto inicial y la hipótesis de inducción de la función auxiliar, al teorema de corrección de la función auxiliar en el cuadro inicial y en aquellas partes de la prueba donde se use el teorema de corrección o la hipótesis de inducción de la función. En resumen, puede necesitarse que la especificación exacta de la función auxiliar se construya incrementalmente en varios intentos.

Segundo, la táctica anterior se ha expuesto en términos de un objetivo con el formato **S[t,z]**. En general este objetivo contiene varios términos y variables. La especificación auxiliar se obtiene de manera que cada término se convierte en una variable de entrada distinta y cada variable de salida en una

función auxiliar diferente.

Esta regla general, tomada de [MaWa87, Traugott89], generaliza lo que en [MaWa80] se consideraban dos estrategias distintas: la formación de funciones auxiliares y la generalización de funciones.

GARANTIA DE TRANSPARENCIA CONSULTIVA

Una propiedad esencial de los programas funcionales es la transparencia consultiva. Una condición necesaria para garantizar la transparencia consultiva de una función es que las únicas variables utilizadas sean sus parámetros, es decir, que no utilice variables globales. En caso contrario, dados unos parámetros, la función podría devolver resultados distintos según el valor de las variables globales.

La programación funcional con Lisp es permisiva en el uso de variables globales, determinándose el valor de éstas mediante una regla de ámbito [WiHo84]. Dado que el método de Manna y Waldinger está orientado a Lisp, no es de extrañar que permita la derivación de funciones con variables globales. Así, en [MaWa87] se deriva la siguiente función **raizcuad**, que calcula la raíz cuadrada de un número real **r** con margen de error **e**:

```
raizcuad (r,e) <= raizcuad2(e) ;
raizcuad2 (e) <= if max(r,1) < e
                 then 0
                 else if (raizcuad2(2*e)+e)^2 ≤ r
                      then raizcuad2(2*e)+e
                      else raizcuad2(2*e) ;
```

donde la función auxiliar **raizcuad2** cita una variable **r** no local que es un parámetro de la función principal **raizcuad**.

Asimismo [Traugott89] deriva la función **ordrapida** de ordenamiento rápido de una lista con la ayuda de dos funciones

auxiliares **menores** y **mayores**, en cuyo cuerpo se cita al parámetro lista de la función **ordrapida**.

Obsérvese que una función auxiliar consulta variables globales para referirse a valores que no varían de llamada en llamada. El modo obvio de eliminar estas consultas es aumentar el número de parámetros de la función con estas variables. Esto significa que la sentencia de corrección de una función no debe contener variables libres ni constantes o funciones de Skolem. Esta restricción obliga a modificar ligeramente la heurística de introducción de funciones auxiliares.

Definición (táctica de identificación de funciones auxiliares)

Supóngase que durante la derivación de una función **f** se obtiene un objetivo:

O[S[t[a],z]]

y posteriormente otro objetivo que contiene una réplica de una parte del anterior:

O'[S[t'[a],z']]

Puede resultar conveniente utilizar una función auxiliar **f'** con condición de salida:

S[b,z]

La nueva condición **S** de salida se obtiene comparando las dos réplicas de **S** contenidas en los objetivos **O** y **O'**. Deben encontrarse pares correspondientes de términos básicos y de variables. Los términos básicos sólo pueden contener símbolos de Skolem correspondientes a los parámetros de la función principal. Asimismo las variables deben aparecer en los términos de salida del objetivo correspondiente. Este conjunto de

pares sirve para determinar las funciones auxiliares y sus parámetros:

- Cada par de términos básicos identifica un parámetro de las funciones auxiliares.
- Cada par de variables identifica una nueva función auxiliar.

Además, todas las funciones auxiliares tienen la misma lista de parámetros. ∎

La restricción sobre las variables evita la derivación de funciones no citadas en el término de salida de la función principal y por tanto sin utilidad para ésta.

La elección de los pares de términos básicos es más difícil, ya que puede haber varias posibilidades de emparejamiento. Como criterio principal se intenta que el número de pares sea el menor posible, y después que los términos sean lo más grande posible. En el criterio de Manna y Waldinger el conjunto de pares de términos se restringe a pares de términos distintos. De esta manera se reduce el número de parámetros de las funciones auxiliares. Si hay algún término básico igual en las dos réplicas, quedan constantes del cuadro en la especificación de las funciones auxiliares; estas constantes corresponden a parámetros de la función principal que se utilizan como variables globales. Al suprimir la restricción de que los términos sean distintos, eliminamos la posibilidad de que algún parámetro se tome como variable global.

Nuestro distinto criterio sobre los términos puede producir diferencias en las deducciones e incluso en los programas derivados. Veámoslo para los dos ejemplos antes citados.

Ejemplo (ordrapida)

Jonathan Traugott ha derivado varios algoritmos de ordena-

ción de listas, que expone con cierto detalle en [Traugott89]. En particular incluye la derivación completa del algoritmo de ordenación rápida y explica la motivación para introducir dos funciones auxiliares, **menores** y **mayores**.

En un momento dado de la derivación dicho autor obtiene el objetivo:

	perm(cola(b),w_1<>w_2)) and if fϵw_1 then f$\leq$cabeza(b) and if gϵw_1 and hϵw_2 then gϵh and if hϵw_2 then cabeza(b)$\leq$h and perm(cola(b),x_1<>w_1) and perm(cola(b),x_2<>w_2)	t

donde t es cierto término de salida.

Si proseguimos la derivación, podemos obtener (tras corregir las erratas del artículo):

	⌜ perm(cola(cola(b)),w_1<>w_2')) and ⌝ ⌊ if fϵw_1 then f$\leq$cabeza(b) and if gϵw_1 and hϵw_2' then gϵh and if hϵw_2' then cabeza(b)$\leq$h and perm(cola(cola(b)),x_1'<>w_1) and perm(cola(cola(b)),x_2<>w_2') and ⌋ if gϵw_1 then g$\leq$cabeza(cola(b)) and not cola(b)=nil and cabeza(b)$\leq$cabeza(cola(b))	t'

El primer objetivo y la parte del segundo encerrada entre corchetes son dos réplicas. Si representamos el primer objetivo, tomando sólo los términos y variables de interés según las restricciones dadas en la táctica, como

O[cabeza(b), cola(b), w_1, w_2],

el segundo tiene la forma

O'[O[cabeza(b), cola(cola(b)), w_1, w_2']]

En la notación anterior, los dos primeros términos son básicos mientras que los dos siguientes son variables. El primer término básico coincide en ambas réplicas, razón por la que Traugott no lo incluyó como parámetro. Nosotros sí lo hacemos, aunque la elección de **cabeza(b)** en lugar de **b** puede ser polémica. Dado que no hay ninguna aparición más de **b**, resulta más lógico tomar **b** en el contexto específico en que se usa; esto permite obtener una especificación más clara de la función auxiliar y una función auxiliar más eficiente. Las dos variables que aparecen en el término de salida dan lugar a las funciones auxiliares **menores** y **mayores**.

El programa derivado por Traugott es:

```
ordrapida (l)
   <= if l=nil
      then nil
      else ordrapida(menores(cola(l))) <>
           cabeza(l)::ordrapida(mayores(cola(l))) ;

<menores(s), mayores(s)>
   <= if s=nil
      then <nil, nil>
      else if cabeza(s)≤cabeza(l)
           then <cabeza(s)<>menores(cola(s)),
                 mayores(cola(b))>
           else <menores(cola(s)),
                 cabeza(s)<>mayores(cola(s))> ;
```

Utilizando nuestro criterio la función derivada es:

```
ordrapida (l)
   <= if l=nil
      then nil
      else ordrapida(menores(cabeza(l),cola(l))) <>
        cabeza(l)::ordrapida(mayores(cabeza(l),cola(l))) ;
```

```
<menores(x,s), mayores(x,s)>
   <= if s=nil
      then <nil, nil>
      else if cabeza(s)≤x
           then <cabeza(s)<>menores(x,cola(s)),
                 mayores(x,cola(b))>
           else <menores(x,cola(s)),
                 cabeza(s)<>mayores(x,cola(s))> ;
```

Puede observarse el aumento de un parámetro en las funciones auxiliares y la ausencia de variables globales. El uso de **x** en vez de la lista **l** completa dentro de las funciones auxiliares logra una pequeña ganancia de eficiencia, al evitarse el cálculo continuo del término **cabeza(l)**.

Por supuesto, en nuestro sistema con patrones el programa derivado es distinto, puesto que define **ordrapida** mediante dos ecuaciones de recurrencia y otras dos para la función auxiliar. Esta es una función única **menoresymayores** de rango tupla (un par de listas), a diferencia de las dos funciones independientes que deriva Traugott, lo que redunda en una computación más eficiente. La extracción de las dos listas obtenidas por cada evaluación de **menoresymayores** se realiza con definiciones locales que evitan el uso de funciones selectoras. El programa derivado es:

```
dec ordrapida : lista -> lista ;
--- ordrapida nil
       <= nil ;
--- ordrapida x::l
       <= let (p,s) == menoresymayores (x,l)
          in ordrapida p<>x::ordrapida s ;

dec menoresymayores : entero#lista -> lista#lista ;
--- menoresymayores (_,nil)
       <= (nil,nil) ;
--- menoresymayores (x,y::l)
       <= let (p,s) == menoresymayores (x,l)
          in if y≤x
             then (y::p,s)
             else (p,y::s) ;
```

Ejemplo (raizcuad)

El caso tomado de [MaWa87] es algo más problemático, porque el nuevo criterio hace que la función auxiliar **raizcuad2** carezca de sentido y que deba cambiarse el orden bien fundado elegido para la inducción.

Supongamos que se ha definido una teoría de números reales. Se trata de desarrollar una función **raizcuad** que, dado un número real no negativo **r** y un margen de error (real) positivo **e**, determine una aproximación a la raíz cuadrada exacta de **r** con un margen de error menor de **e** y que sea menor o igual que la raíz exacta. Formalmente:

raizcuad (r,e) <= encontrar z
tal que $z^2 \leq r$ and not $(z+e)^2 \leq r$
donde $0 \leq r$ and $0 < e$

La derivación se hace por inducción bien fundada. El cuadro inicial contiene las dos filas:

asertos	objetivos	raizcuad(r,e)
1. $0 \leq r$ and $0 < e$		
	2. $z^2 \leq r$ and not $(z+e)^2 \leq r$	z

Resolviendo el objetivo consigo mismo (más concretamente el primer conjuntado con el segundo) y simplificando **e+e** como **2*e** obtenemos:

	3. $z'^2 \leq r$ and not $(z'+2*e)^2 \leq r$	if $(z'+e)^2 \leq r$ then z'+e else z'

Podemos representar los objetivos 2 y 3 como:

```
O[e, z]
O[2*e, z']
```

Las variables **z** llevaban a Manna y Waldinger a considerar la conveniencia de una función auxiliar, y el término discrepante a determinar que tuviera un solo parámetro. El programa final tiene el formato:

```
raizcuad (r,e)  <=  raizcuad2(e) ;
raizcuad2 (e)   <=  t[e,r] ;
```

donde **t** es un término funcional.

Nosotros obligamos a que no haya variables libres, así que el conjunto de pares de términos correspondientes incluye términos iguales. Podemos representar los objetivos 2 y 3 de la forma:

```
O[r, e, z]
O[r, 2*e, z']
```

El resultado es que la función auxiliar tiene dos parámetros, obteniéndose el programa de esquema:

```
raizcuad (r,e)  <=  raizcuad2(r,e) ;
raizcuad2 (r,e)  <=  t[e,r] ;
```

donde la función auxiliar carece de sentido porque hace lo mismo (obviamente también tiene la misma especificación) que la función principal.

La derivación de la función auxiliar como Manna y Waldinger tiene una ventaja en la elección del orden bien fundado de la inducción. La relación elegida es la relación de duplicamiento acotado $<_{da(y)}$ definido sobre reales positivos de forma que:

$$u <_{da(y)} v \equiv u=2*v \text{ and } v \leq y$$

para algún límite superior prefijado **y**. Este límite superior

es un parámetro más de la relación, puesto que para cada número real **y** se obtiene una relación diferente $<_{da(y)}$. La introducción como un aserto de la definición de la relación de duplicamiento acotado deja a **y** como una variable, cuyo valor no necesita fijarse hasta más adelante. Esta flexibilidad en la elección del límite superior necesita que **r** sea una variable global, como vemos a continuación.

La relación bien fundada << concreta utilizada en la hipótesis de inducción ($<_w$ en el sistema de Manna y Waldinger) no necesita especificarse hasta avanzada la deducción. Si se consideran las relaciones bien fundadas como objetos del dominio (en una lógica monogénero), la proposición $x<_w y$ puede considerarse como una abreviatura de **<(w,x,y)**. En consecuencia el principio de inducción bien fundada está cuantificado sobre todas las relaciones, es decir, está rodeado por el cuantificador **(V w)**. Al eliminar cuantificadores por escolemización, la entrada **a** del programa se convierte, no en una constante **a**, sino en un término **a(w)**. Esto hace que la relación **w** elegida no puede depender de **a** porque **w** no puede unificarse con ningún término que contenga **a(w)**.

Si tomamos $<_{da(y)}$ como relación $<_w$, para algún **y**, **w** se instancia a **da(y)**. Tomando ahora **max(r,1)** como valor de **y**, **w** se instancia a **da(max(r,1))**. Esto sólo es posible si **r** es una constante; si fuera un parámetro estaríamos intentando instanciar **w** a **da(max(r(w),1))**. La consideración de **r** como constante se consigue dejando **r** como una variable global del programa.

Nuestro criterio nos impide definir una función auxiliar, pero también hace que no podamos usar $<_{da(y)}$ como orden bien fundado por la circularidad de **w** descrita en el párrafo anterior. Evitamos la circularidad en la relación $<_w$ tomando una relación menos general. El nuevo orden relaciona pares de reales, ya que la inducción se hace sobre los parámetros de **raizcuad**:

$(u,v) <_{dal} (r,s) \equiv u=r \text{ and } v=2*s \text{ and } s \leq \max(u,1)$

La nueva relación es mucho más específica en su límite superior y menos útil en general. Sin embargo, ya no depende de ningún parámetro **y**; por tanto, aunque **r** represente **r(w)**, **w** sólo se instancia a **dal**.

La derivación de **raizcuad** con el nuevo criterio y orden bien fundado produce el programa (con definiciones locales):

```
raizcuad (r,e)
   <= if max(r,1)<e
      then 0
      else let v == raizcuad(r,2*e)
           in if (v+e)^2≤r then v+e else v ;
```

La derivación también puede hacerse en nuestro sistema, pero necesitamos suponer que el tipo de los reales es un tipo predefinido (ya que no se declara como un álgebra de palabras) y que disponemos de una teoría de números reales. La deducción y el programa final son prácticamente iguales. ∎

4. EJEMPLOS DE APLICACION

El sistema deductivo definido en el Capítulo 3 permite derivar funciones basadas en patrones. En este capítulo aplicamos el sistema a diversos problemas de programación. La selección de los problemas se ha realizado de manera que pueda verse el uso de las diferentes técnicas desarrolladas. También se ha querido usar cierta variedad de tipos de datos, eligiendo los enteros, las listas de enteros y los conjuntos de enteros. A continuación damos la relación de programas incluidos, junto con los aspectos más interesantes de cada derivación:

(i) Máximo común divisor, mediante restas sucesivas. Se razona por inducción bien fundada seguida de descomposición de un par de enteros en cuatro casos.

(ii) Inversión de una lista. A diferencia de los demás ejemplos, no realiza una síntesis de función sino una optimización. Se necesita una función auxiliar, derivada mediante inducción estructural de dos casos.

(iii) Máximo elemento de una lista. Se utiliza inducción estructural de tres casos, uno de los cuales no deriva ninguna ecuación. La ecuación derivada en el caso recursivo contiene una definición local y variables anónimas y sinónimas.

(iv) Ordenación de una lista por selección ("select-sort"). Se utiliza inducción bien fundada, descomponiendo el parámetro en dos casos. En el caso de lista no vacía la ecuación derivada puede contener una definición local con patrones y variables anónimas y sinónimas; además, la demostración de su especificación parcial precisa un lema auxiliar, que

sirve como especificación de una función de selección.

La función auxiliar se demuestra por inducción estructural de tres casos, uno de ellos sin efecto derivador y en el caso recursivo derivando patrones locales, más la posibilidad de derivar variables anónimas y sinónimas.

(v) Intersección de dos conjuntos. Se usa el principio de inducción estructural de conjuntos con dos casos.

Las funciones con patrones derivadas son una muestra de las funciones derivadas por el autor. A lo largo de la tesis se incluyen ejemplos de algún paso de la derivación de estas funciones. Sin embargo, es un sinsentido incluir la derivación de todas ellas, por lo que simplemente damos la relación completa:

- Otras versiones del máximo común divisor, variando en especificación o programa.
- Frente y último elemento de una lista, como dos funciones independientes y como una sola.
- Otros tres algoritmos de ordenación de listas, con sus funciones auxiliares: ordenación rápida ("quicksort"), por mezcla ("mergesort") y por inserción ("insertsort").
- Operaciones sobre conjuntos: borrado de un elemento de un conjunto y unión, diferencia y producto cartesiano de dos conjuntos (esta última con el producto de un entero por un conjunto como función auxiliar).
- Optimización de una función que aplana un árbol en una lista.

4.1. MAXIMO COMUN DIVISOR

Deseamos derivar una función que halle el máximo común divisor de dos enteros no negativos. Seguimos los pasos des-

critos en el apartado 3.9.

Teoría lógica

Supongamos que disponemos de la teoría combinada de los tipos predefinidos en miniHope, que incluye una teoría de los enteros no negativos. Para mayor comodidad incluimos la lista de axiomas y teoremas de la teoría que necesitaremos durante la deducción. Las sentencias están escolemizadas, tal como aparecerían en el cuadro deductivo en forma de asertos. En concreto las variables universales aparecen como variables libres. Al lado de cada sentencia indicamos el nombre dado a la misma, para su fácil localización.

$$\text{succ } x>0 \qquad (>\text{-cero})$$

$$x>0 \text{ and } y\geq x \equiv y-x<y \qquad (>\text{-resta})$$

$$x|x \qquad (|\text{-reflexiva})$$

$$x|0 \qquad (|\text{-cero})$$

$$\text{if } y\geq x \text{ then } \left[\begin{array}{c} z|x \text{ and } z|y \\ \equiv \\ z|x \text{ and } z|(y-x) \end{array}\right] \qquad (|\text{-resta})$$

$$(x1,x2)<_{\text{prog}(<,<)}(x3,x4) \equiv \left[\begin{array}{c} x1=x3 \text{ and } x2<x4 \\ \text{or} \\ x1<x3 \text{ and } x2=x4 \end{array}\right](<_{\text{prog}(<,<)})$$

También disponemos de la propiedad de totalidad de la relación ≥, que sólo es utilizable en resolució por teoría:

$$x\geq y \text{ or } y\geq x \qquad (\geq\text{-totalidad})$$

Especificación de función

De una manera informal, el máximo común divisor de dos enteros no negativos es un entero no negativo tal que es el mayor divisor de todos los divisores posibles de los dos enteros primeros. Algo más sutil es determinar qué quiere decir "entero mayor", es decir, determinar la relación de orden entre enteros que debe tomarse para determinar de manera única el máximo común divisor. La interpretación más corriente es

tomar la conocida relación ≤ entre enteros. De esta manera la especificación de la función **mcd** que calcula el máximo común divisor de dos enteros queda:

```
mcd (x,y):entero#entero
     <=  encontrar  z:entero
         tal que  z|x and z|y and
                  (∀ w:entero) if w|x and w|y then w≤z
     donde  x>0 or y>0
```

Obsérvese que la condición de entrada obliga a que al menos uno de los dos parámetros sea mayor que 0. Si no fuera así, no habría máximo común divisor porque cualquier entero es divisor de 0 y el número de enteros es infinito.

Podemos idear otra especificación en que la relación "mayor" sea la relación |. Es decir, un entero será menor que otro cuando sea su divisor. Con esta nueva interpretación no se necesita condición de entrada, y por tanto no se restringen los posibles valores de los parámetros. La nueva especificación es:

```
mcd (x,y):entero#entero
     <=  encontrar  z:entero
         tal que  z|x and z|y and
                  (∀ w:entero) if w|x and w|y then w|z
```

Adoptamos esta especificación, por ser menos corriente y por cubrir más valores de entrada. En todo caso se deriva la siguiente declaración de tipo de la función:

```
dec mcd : entero#entero -> entero ;
```

Recordemos que la especificación es un endulzamiento de la especificación verdadera, que es:

```
(∀ x,y:entero) [ z|x  and  z|y  and
(∃ z:entero)   [ (∀ w:entero) if w|x and w|y then w|z ]
```

Admitimos cualquier función de la teoría como función primitiva.

Formación del cuadro inicial

Posiblemente haya varias alternativas sobre el modo de demostrar la especificación. Sin embargo, la relación "divisor" no parece ser proclive a utilizar inducción estructural, por lo que usamos inducción bien fundada. La sentencia inductiva por demostrar es:

$$(\forall\ x,y{:}\text{entero})(\exists\ z{:}\text{entero})\left[\begin{array}{l}\text{if } \left[\begin{array}{l}\text{if } (u,v)\ll(x,y)\\ \text{then } \left[\begin{array}{l}\text{mcd}(u,v)\,|\,u \text{ and } \text{mcd}(u,v)\,|\,v \text{ and}\\ (\forall\ w{:}\text{entero})\ \text{if } w|u \text{ and } w|v\\ \qquad\qquad\qquad\qquad \text{then } w|\text{mcd}(u,v)\end{array}\right]\end{array}\right]\\ \text{then } \left[\begin{array}{l}z|x \text{ and } z|y \text{ and}\\ (\forall\ w{:}\text{entero})\ \text{if } w|x \text{ and } w|y\\ \qquad\qquad\qquad\qquad \text{then } w|z\end{array}\right]\end{array}\right]$$

Asimismo, esta sentencia puede demostrarse por descomposición sobre pares de enteros. Distinguimos cuatro casos posibles de valores de los dos parámetros, según sea cero o positivo cada uno. La sentencia resultante es una conjunción de cuatro subsentencias cerradas, que corresponden a cuatro especificaciones parciales. Cada especificación parcial se demuestra por separado con un cuadro parcial recursivo. No incluimos aquí la nueva sentencia, sino que directamente formamos cada cuadro parcial inicial.

Dos enteros ceros

El cuadro parcial recursivo inicial es:

asertos	objetivos	mcd (0,0)
1. if (u,v)<<(0,0) then mcd(u,v)|0 and if n|0 then n|mcd(u,v)		
	2. $\boxed{z}\,|\,\boxed{0}^{+}$ and if w|0 then w|z	z

donde **u**, **v**, **n**, **z** son variables y **w** es una constante de Skolem.

Debe destacarse que mientras en el objetivo **w** es una constante de Skolem, su análogo **n** en la hipótesis de inducción es una variable. Esto se debe a la distinta fuerza de ambos cuantificadores en el cuadro y, por tanto, a que se eliminan con distintas reglas de escolemización. Obsérvese que han sido automáticamente simplificadas las subsentencias **mcd(u,v)|0 and mcd(u,v)|0** y **n|0 and n|0** de la hipótesis de inducción y **z|0 and z|0** y **w|0 and w|0** del objetivo inicial.

Por el axioma **|-cero** sabemos que cualquier entero es divisor de 0, por lo que podemos eliminar del objetivo el primer conjuntado. De manera formal, resolvemos la subsentencia encuadrada del objetivo con el axioma **|-cero**:

$\boxed{x|0}^-$

con unificador (más general) {x<-z}, resultando:

	3. if w|0 then w|z	z

que por partición se transforma en las dos filas:

4. w|0		
	5. $\boxed{w|z}^+$	z

Ahora nos falta demostrar que existe un entero **z** tal que un entero **w** cualquiera es divisible entre él, pero sabemos que esta condición la satisface el 0. Resolviendo por segunda vez el objetivo con el axioma **|-cero**, con unificador {z<-0}, obtenemos el objetivo:

	6. true	0

Tras este paso deductivo hemos obtenido el objetivo **true**,

teniendo un cuadro obviamente cierto. Así pues, formamos la ecuación correspondiente al cuadro:

```
--- mcd (0,0)
        <= 0 ;
```

Un entero cero

Veamos el caso del primer parámetro cero y el segundo positivo. La hipótesis de inducción y el objetivo inicial no se simplifican como en el caso anterior.

asertos	objetivos	mcd (0,succ y)
1. if (u,v)<<(0,succ y) then mcd(u,v)|u and mcd(u,v)|v and if n|u and n|v then n|mcd(u,v)		
	2. z|0 and z|succ y and if w|0 and w|succ y then w|z	z

Igual que en el caso anterior, el conjuntado que afirma que la salida es divisible por cero es cierto. Realizando resolución con el axioma **|-cero**, obtenemos el nuevo objetivo:

	3. $\boxed{z\|succ\ y}^{+}$ and if w|0 and w|succ y then w|z	z

En este punto de la deducción debemos encontrar un entero **z** que sea divisor del entero positivo **succ y**. Además cualquier otro entero **w** divisor de **succ y** también debe serlo de **z**. Podemos satisfacer ambas condiciones si tomamos el mismo **succ y** como **z**, ya que todo entero es el mayor divisor de sí mismo. Para este razonamiento necesitamos el axioma **|-reflexiva**:

$$\boxed{x|x}^{-}$$

Resolvemos el objetivo 3 con este axioma, con unificador {x<-succ y, z<-succ y}.

	4. if w|0 and w|succ y then w|succ y	succ y

donde el término de salida ya es un término básico.

Por partición el objetivo se descompone en las filas:

5. w|0		
6. w|succ y		
	7. w|succ y	succ y

Resolviendo las filas 6 y 7 obtenemos el cuadro final:

	8. true	succ y

En este momento formamos la ecuación derivada. Sin embargo, debe notarse que el término de salida cita un subtérmino, no variable, del patrón de la ecuación. Por tanto, formamos la ecuación señalando que el subtérmino **succ y** tenga el sinónimo **n**. La ecuación resultante es:

```
--- mcd (0,n & succ _)
        <= n ;
```

La tercera especificación parcial se demuestra exactamente igual que la segunda sólo que cambiando el papel de los parámetros. Por tanto obtenemos la ecuación:

```
--- mcd (m & succ _,0)
        <= m ;
```

Dos enteros no ceros

La cuarta y última especificación parcial tiene los dos parámetros distintos de cero, resultando en el cuadro parcial

recursivo inicial:

asertos	objetivos	mcd(succ x,succ y)
1. if (u,v)<<(succ x,succ y) then mcd(u,v)|u and mcd(u,v)|v and if n|u and n|v then n|mcd(u,v)		
	2. z|succ x and z|succ y and if w|succ x and w|succ y then w|z	z

El objetivo afirma que existe un entero **z** divisor de **succ x** y **succ y** y múltiplo de cualquier otro divisor de estos números. Según el axioma **|-resta**, sabemos que si dicho entero existe y **succ y** es mayor o igual que **succ x**, también es un divisor de **succ x** y **succ y - succ x**:

if y≥x then $z|x \text{ and } z|y \equiv z|x \text{ and } z|y-x$ -

En el cuadro aplicamos reemplazamiento por equivalencia entre los dos conjuntados primeros del objetivo y el consecuente de este axioma. La regla puede aplicarse porque el consecuente del aserto tiene polaridad negativa en el cuadro. El objetivo resultante es:

	3. z|succ x and z|(succ y - succ x) and if w|succ x and w|succ y then w|z and succ y≥succ x	z

donde aparece una nueva condición a satisfacer, la que ha permitido realizar el razonamiento del axioma **|-resta**, es decir, que el segundo parámetro de la ecuación es mayor que el primero.

Reemplazando por equivalencia de nuevo entre el axioma **|-resta** y el antecedente de la implicación contenida en el objetivo, obtenemos:

	4. z|succ x and z|(succ y - succ x) and [if w|succ x and w|(succ y - succ x) [then w|z and succ y≥succ x	z

Obsérvese que los tres primeros conjuntados del último objetivo son una instancia del consecuente de la hipótesis de inducción, tomando como unificador {u<-succ x, v<-succ y-succ x, z<-mcd(succ x,succ y - succ x), n<-w}. Resolviendo ambas sentencias obtenemos:

	5. succ y≥succ x and $\boxed{\text{(succ x,succ y-succ x)}\ll\text{(succ x,succ y)}}^{+}$	mcd(succ x,succ y-succ x)

donde el término de salida ya es un término básico. El objetivo resultante es cierto si demostramos la condición sobre el valor relativo de los parámetros más la condición de aplicación del paso inductivo. Esta última condición garantiza la terminación de la función derivada, declarando que los parámetros de la llamada recursiva son menores que los originales según algún orden bien fundado <<. Tomamos la relación progresiva $<_{\mathbf{prog}}$ (sobre < y <) definida por el axioma:

$$\boxed{(x1,x2) <_{prog(<,<)} (x3,x4)}^{\pm} \equiv \left[\begin{array}{c} x1=x3 \text{ and } x2<x4 \\ \text{or} \\ x1<x3 \text{ and } x2=x4 \end{array}\right]$$

Resolviendo el objetivo con este axioma, bajo unificador {x1<-succ x, x2<-succ y-succ x, x3<-succ x, x4<-succ y}, obtenemos:

	6. succ y ≥ succ x and [$\boxed{\text{succ y-succ x < succ y}}^{+}$ or [succ x<succ x and succ y-succ x = succ y	mcd (succ x,succ y-succ x)

Debe notarse que, además de las usuales simplificaciones

true-false, se ha simplificado la igualdad **succ x=succ x** que, según el axioma $<_{prog(<,<)}$, debería aparecer en el primer disjuntado del conjuntado introducido.

El segundo conjuntado puede demostrarse fácilmente demostrando la verdad de su primer disjuntado. Para ello debe observarse que todo entero reduce su valor (según <) si le restamos un entero positivo menor, como dice el axioma **>-resta:**

$$x>0 \text{ and } y\geq x \equiv \boxed{y-x<y}^{-}$$

Podemos resolver el primer disjuntado del segundo conjuntado y el axioma **>-resta**, obteniendo el objetivo:

	7. succ y ≥ succ x and succ x > 0	mcd (succ x,succ y-succ x)

El objetivo resultante puede resolverse con el axioma **>-cero:**

succ x>0

creándose el objetivo siguiente:

	8. succ y ≥ succ x	mcd(succ x, succ y-succ x)

En este punto de la demostración tenemos que, si demostramos que el segundo parámetro es mayor que el primero, el término funcional:

mcd (succ x,succ y-succ x)

es un término que satisface la especificación. Desgraciadamente, la única restricción que deben cumplir los parámetros es que ambos sean positivos, sin poder afirmar más sobre su

valor relativo. Por tanto, debemos dejar de momento esta rama de la demostración. Si derivamos otro término con un objetivo asociado complementario del actual, podremos terminar la deducción con éxito.

Efectivamente, podemos realizar dicha rama deductiva, igual que la anterior, sólo que con los parámetros de la ecuación jugando papeles simétricos. En resumen, si aplicamos sucesivamente:

- reemplazamiento por equivalencia con |-resta,
- reemplazamiento por equivalencia con |-resta,
- resolución con la hipótesis de inducción,
- resolución con $<_{prog(<,<)}$,
- resolución con >-resta,
- resolución con >-cero,

a partir del objetivo 2 obtenemos la sucesión de objetivos siguientes:

	9. z|(succ x - succ y) and z|succ y and if w|succ x and w|succ y then w|z and succ x≥succ y	z
	10. z|(succ x - succ y) and z|succ y and [if w|(succ x - succ y) and w|succ x then w|z] and succ x≥succ y	z
	11. succ x≥succ y and (succ x-succ y,succ y)$<_{prog}$(succ x,succ y)	mcd (succ x-succ y,succ y)
	12. succ x≥succ y and [succ x-succ y=succ x and succ y<succ y or succ x-succ y<succ x]	mcd (succ x-succ y,succ y)
	13. succ x≥succ y and succ y>0	mcd (succ x-succ y,succ y)
	14. succ x ≥ succ y	mcd (succ x-succ y,succ y)

Podemos aplicar resolución con teoría entre los objetivos

8 y 14, invocando el teorema **≥-totalidad:**

x≥y or y≥x

La aplicación de esta regla produce el objetivo:

	14. true	if succ x ≥ succ y then mcd(succ x-succ y,succ y) else mcd(succ x,succ y-succ x)

El objetivo obtenido es obviamente cierto, así que la demostración de la especificación parcial ha concluido. Sin embargo, como en el caso anterior, podemos formar la ecuación derivada con dos variables **m**, **n** sinónimas respectivamente de los términos **succ x**, **succ y**, resultando:

```
--- mcd (m & succ _, n & succ _)
       <= if m≥n
          then mcd (m-n, n)
          else mcd (m, n-m) ;
```

Ampliación de la teoría

Una vez derivada la función **mcd**, podemos ampliar la teoría combinada existente con los axiomas computacionales que definen la función:

```
mcd (0,0) = 0
(∀ x:entero)  mcd (0,succ x) = succ x
(∀ y:entero)  mcd (succ y,0) = succ y
                ┌ mcd (succ x,succ y)                                  ┐
                │              =                                       │
(∀ x,y:entero)  │ ┌ if succ x ≥ succ y                                 ┐ │
                │ │ then mcd (succ x - succ y, succ y)                 │ │
                └ └ else mcd (succ x, succ y - succ x)                 ┘ ┘
```

más el teorema de corrección del máximo común divisor:

$$(\forall\ x,y{:}entero)\ \left[\begin{array}{l} mcd(x,y)\,|\,x \text{ and } mcd(x,y)\,|\,y \text{ and} \\ (\forall\ w{:}entero)\ \left[\begin{array}{l} \text{if } w\,|\,x \text{ and } w\,|\,y \\ \text{then } w\,|\,mcd(x,y) \end{array}\right] \end{array}\right]$$

En sucesivos ejemplos no detallamos tanto los pasos realizados. No distinguimos explícitamente las distintas etapas de la derivación, que incluso se intercambian cuando su orden es irrelevante.

4.2. INVERSION DE UNA LISTA

Se trata de un problema ampliamente conocido en programación transformativa. Sea la función de inversión de listas:

```
dec invertir : lista -> lista ;
--- invertir nil
       <= nil ;
--- invertir x::l
       <= invertir l <> (x::nil) ;
```

Si la única función de listas predefinida en el intérprete del lenguaje es el constructor ::, es decir, si <> debe definirse recursivamente, la función anterior es ineficiente porque su complejidad en el número de "cons" realizados es cuadrática sobre la longitud de la lista. Sin embargo parece razonable esperar que la operación se pueda hacer con una complejidad lineal.

El problema consiste en obtener un programa eficiente de inversión de listas mediante transformación del programa anterior.

La especificación de nuestra función es:

```
invertir l:lista  <=  encontrar  z:lista
                      tal que  z = invertir l
```

que deriva una declaración de tipo como la ya existente.

La derivación de la función auxiliar va a precisar de un conjunto de axiomas que incluimos a continuación.

```
l = l                                              (=-reflexiva)
nil <> l = l                                       (<>-nil-izdo)
l <> nil = l                                       (<>-nil-dcho)
x::nil <> l = x::l                                 (<>-átomo-izdo)
(l1 <> l2) <> l3 = l1 <> (l2 <> l3)                (<>-asociativa)
invertir nil = nil                                 (invertir-nil)
invertir x::l = invertir l <> x::nil               (invertir-::)
```

Consideramos primitivas todas las funciones, excepto **invertir** (ya que entonces obtendríamos, por reflexividad de la igualdad, el programa sin sentido **invertir l = invertir l**) y <> (ya que es una operación cara de implementar).

FUNCION PRINCIPAL

El cuadro inicial es:

asertos	objetivos	invertir l
	1. z = invertir [l]	z

Podemos conseguir la versión eficiente si disponemos de una función auxiliar **inv**, definida con la sentencia (escolemizada) de corrección:

[inv (l1,l2) = invertir l1 <> l2] -

Dicha función no se encuentra disponible, por lo que se interrumpe la derivación de **invertir** y se comienza la derivación de **inv**. En programación funcional, el cometido se describe diciendo que se construye una función auxiliar con un parámetro acumulador adicional, donde ir guardando el resultado. En demostración de teoremas la tarea consiste en demos-

trar una sentencia más general que la original. En todo caso, no analizamos la motivación por la que se decide derivar dicha función, ya que se ha descrito abundantemente en otros textos [BuDa77, MaWa79, MaWa80, Scherlis81].

Para no interrumpir aquí la derivación de **invertir**, supongamos que la función auxiliar **inv** se ha derivado satisfactoriamente. La teoría se incrementa con los axiomas computacionales y la sentencia de corrección de la nueva función. El cuadro correspondiente a **invertir** también se incrementa con estas sentencias.

Podemos realizar reemplazamiento por igualdad entre el lado derecho de la igualdad del objetivo inicial y el axioma **<>-nil-dcho:**

[l <> nil = l]$^{-}$

Previamente deben redenominarse las variables de alguna de las dos sentencias, ya que se da el mismo símbolo a la constante **l** del objetivo y a la variable **l** del axioma; p.ej. podemos redenominar la variable **l** del axioma **<>-nil-dcho** como **s**. Aplicando reemplazamiento por igualdad con sustitución {s<-l}, obtenemos el nuevo objetivo:

	2. z = [invertir (<> nil]	z

Ahora el lado derecho del segundo objetivo es una instancia del lado derecho de la sentencia de corrección de **inv**, tomando {l1<-l, l2<-nil}. Aplicando reemplazamiento por igualdad entre ambas sentencias, obtenemos:

	3. [z = inv (l,nil)]$^{+}$	z

Por último, resolvemos este objetivo con el axioma de

igualdad de listas (renombrando su variable):

| s = s | -

con unificador {z<-inv(l,nil), s<-inv(l,nil)},

	4. true	inv (l,nil)

El cuadro anterior es un cuadro final, así que formamos la ecuación correspondiente:

```
--- invertir l
       <=  inv (l,nil) ;
```

En lo sucesivo no describirmos tan detalladamente todos los pasos de deducción, p.ej. no escribiendo un aserto o un unificador utilizados.

FUNCION GENERALIZADA

La función auxiliar tiene como especificación:

```
inv (l1,l2):lista#lista  <=  encontrar  z:lista
                              tal que   z = invertir l1<>l2
```

correspondiente al axioma de corrección utilizado durante la derivación de **invertir**. La especificación deriva la declaración de tipo de **inv**:

```
dec inv : lista#lista -> lista ;
```

De nuevo, las funciones **invertir** y **<>** se consideran no primitivas.

Demostramos la nueva sentencia de especificación por inducción estructural sobre listas (sobre **l1**) con dos casos.

Lista vacía

La especificación parcial del caso básico es:

(∀ l2:lista) (∃ z:lista) z = invertir nil <> l2

que tiene asociado el cuadro parcial:

asertos	objetivos	inv (nil,l2)
	1. z = invertir nil <> l2	z

donde **l2** es una constante de Skolem.

Aplicando sucesivamente reemplazamiento por igualdad con los axiomas **invertir-nil** y **<>-nil-izdo**, obtenemos los siguientes objetivos:

	2. z = nil <> l2	z
	3. z = l2	z

En este punto de la deducción sabemos que si encontramos algún valor de **z** igual a **l2**, el término de salida es **z**, lo cual se satisface obviamente haciendo que **z** tome el valor **l2**. Formalmente, aplicamos resolución entre el objetivo 3 y el axioma reflexivo de la igualdad de listas, obteniendo,

	4. true	l2

Este objetivo es el objetivo final, así que damos por concluida la derivación de la primera ecuación de **inv**, que queda:

```
--- inv (nil,l2)
      <= l2 ;
```

Lista general

La segunda especificación parcial de **inv** es:

$$(\forall\ x{:}entero)\ (\forall\ l1{:}lista)\ \left[\begin{array}{l} \text{if } (\forall\ l2{:}lista)\ inv(l1,l2)=invertir\ l1{<>}l2 \\ \text{then } (\forall\ l2{:}lista)\ (\exists\ z{:}lista) \\ \qquad z=invertir(x::l1){<>}l2 \end{array}\right]$$

a la que corresponde el cuadro inicial recursivo (estructural):

asertos	objetivos	inv (x::l1,l2)
inv (l1,l) = invertir l1 <> l		
	1. z = invertir(x::l1) <> l2	z

Debe observarse que el proceso de escolemización deja **x**, **l1** y **l2** como constantes, mientras que **l** y **z** son variables.

El modo de proceder es parecido al caso anterior. Modificamos el lado derecho de la igualdad del objetivo hasta obtener un término formado exclusivamente por funciones primitivas. Vamos a ser breves. El objetivo puede reescribirse sucesivamente aplicando reemplazamiento por igualdad mediante las sentencias **invertir-::**, **<>-asociativa** y **<>-átomo-izdo**, obteniendo la secuencia:

	2. z = (invertir l1 <> x::nil) <> l2	z
	3. z = invertir l1 <> (x::nil <> l2)	z
	4. z = invertir l1 <> x::l2	z

Obsérvese que el lado derecho del último objetivo es una instancia del lado derecho de la hipótesis de inducción, utilizando la sustitución {l<-x::l2}. Por tanto si reemplazamos por igualdad, con este unificador, obtenemos el objetivo:

	5. z = inv (l1,x::l2)	z

Por fin hemos obtenido un término derecho primitivo en la igualdad. Si resolvemos este objetivo con el axioma **=-reflexiva**, resulta el nuevo objetivo:

	6. true	inv (l1,x::l2)

Este objetivo es un objetivo final, así que la ecuación derivada es:

```
--- inv (x::l1,l2)
       <= inv (l1,x::l2) ;
```

En este momento añadiríamos a la teoría, como supusimos durante la derivación de **invertir**, los dos axiomas computacionales de **inv** más su sentencia de corrección.

Resumiendo la derivación completa, hemos obtenido un nuevo programa de inversión de listas. Este programa es menos claro que el programa inicial pero más eficiente:

```
dec invertir : lista -> lista ;
--- invertir l
       <= inv (l,nil) ;

dec inv : lista#lista -> lista ;
--- inv (nil,l2)
       <= l2 ;
--- inv (x::l1,l2)
       <= inv (l1,x::l2) ;
```

4.3. MAXIMO ELEMENTO DE UNA LISTA

Deseamos obtener una función que, dada una lista no vacía,

nos devuelva el elemento máximo de la lista. Podemos encontrar en [Bibel80] otra derivación para este problema. Utilizamos la siguiente especificación:

```
maxlista l:lista  <=  encontrar  z:entero
                      tal que  zєl  and
                               (∀ w:entero) if wєl then w≤z
                      donde  not l=nil
```

que deriva la declaración de tipo:

```
dec maxlista : lista -> entero ;
```

Durante la derivación utilizaremos las siguientes sentencias válidas en la teoría. La primera sentencia trata con enteros, mientras las siguientes tratan con listas de enteros.

```
x ≤ x                                              (≤-reflexiva)
l = l                                              (=-reflexiva)
not xєnil                                          (є-nil)
xєy::l  ≡  x=y or xєl                              (є-::)
xєl1<>l2  ≡  xєl1 or xєl2                          (є-<>)
```

Además disponemos de varias sentencias válidas sobre enteros que sólo podemos utilizar en aplicaciones de resolución por teoría.

```
x≤y  ≡  x=y or x<y                        (<-cierre-reflexivo)
if x≤y and y≤z then x≤z                        (≤-transitiva)
x≤y or y≤x                                     (≤-totalidad)
```

Vamos a derivar la función por inducción estructural (sobre la lista l) con tres casos.

Lista vacía

El cuadro inicial se forma como siempre:

asertos	objetivos	maxlista nil
1. not nil=nil		
	2. z∈nil and if w∈nil then w≤z	z

Este cuadro se satisface de forma inmediata, ya que se produce una contradicción. En efecto, si resolvemos el aserto con el axioma **=-reflexiva**, obtenemos:

3. false		

La fila es un aserto final sin término de salida, por lo que debe tratarse (ver apartado 3.5) de un caso en el que los parámetros no cumplen la condición de entrada (como fácilmente se ve). En consecuencia no derivamos ecuación alguna para esta especificación parcial.

Lista atómica

Sea el cuadro parcial inicial:

asertos	objetivos	maxlista x::nil
	1. z∈x::nil and if w∈x::nil then w≤z	z

donde **z** es una variable, **x** es una constante y **w** representa el término de Skolem **w(x)**.

En el cuadro inicial hemos suprimido el aserto de la condición de entrada puesto que es una instancia del axioma de unicidad de los constructores de listas.

Utilizando dos veces reemplazamiento por equivalencia con el axioma **∈-::** podemos reescribir las proposiciones de pertenencia a una lista. Los sucesivos objetivos son:

	2. (z=x or z∈nil) and if w∈x::nil then w≤z	z
	3. (z=x or z∈nil) and if (w=x or w∈nil) then w≤z	z

Sabemos que ningún elemento pertenece al conjunto vacío; por tanto, el objetivo anterior se simplifica sustituyendo la proposición **w∈nil** por **false**. Aplicamos resolución con la sentencia **∈-nil**, obteniendo el objetivo:

	4. (z=x or z∈nil) and if w=x then w≤z	z

El primer conjuntado del objetivo puede satisfacerse mediante resolución con el axioma reflexivo de los enteros, con unificador {z<-x}, resultando el objetivo:

	5. if w=x then w≤z	x

Obsérvese que el unificador utilizado ha hecho que el término se salida se instancie a la constante **x**. Así pues, si logramos demostrar el objetivo 6, **x** será un buen término de salida para la especificación parcial que nos ocupa.

El objetivo nuevo es claramente cierto tomando **z** igual a **x**. Puesto que el objetivo es una implicación, primero se parte automáticamente, resultando en un aserto y un objetivo:

6. [w=x]⁻		
	7. [w≤z]⁺	x

Ahora podemos resolver por teoría ambas filas. Utilizamos la propiedad **<-cierre-reflexivo** como sentencia invocada

(renombrando la variable **x** a **x'**):

$$\boxed{x' \leq y}^{\pm} \equiv \boxed{x' = y}^{\pm} \text{ or } x' < y$$

con unificador {z<-x, x'<-w, y<-x}. El objetivo deducido es:

	8. true	x

El nuevo objetivo es un objetivo final, así que se da por concluida la derivación, que ha producido la ecuación:

```
--- maxlista x::nil
      <=  x ;
```

Lista general

Nos queda el caso inductivo, en que la lista tiene al menos dos elementos. El cuadro inicial consta de un objetivo más la hipótesis de inducción estructural, que tras escolemización y partición-and queda:

asertos	objetivos	maxlista x::y::l
1. maxlista y::l ϵ y::l		
2. if vϵy::l then v$\leq$maxlista y::l		
	3. zϵx::y::l and if wϵx::y::l then w$\leq$z	z

donde **v**, **z** son variables, **x**, **y**, **l** son constantes y **w** representa el término básico **w(x,y,l)**.

De forma análoga a como hicimos durante la derivación de la ecuación anterior, desarrollamos las proposiciones de pertenencia a una lista no vacía.

Reemplazando por equivalencia dos veces con el axioma ϵ-::, obtenemos la secuencia de objetivos siguiente:

	4. (z=x or z∈y::l) and if w∈x::y::l then w≤z	z
	5. (z=x or z∈y::l) and if (w=x or w∈y::l) then w≤z	z

Examinando el objetivo encontramos una disjunción de dos casos en cada conjuntado. Centrémonos en el segundo conjuntado. La sentencia puede quedar más clara si consideramos por separado los dos casos contemplados en el antecedente. Si **w** es igual a **x** podemos reemplazar en el consecuente **w** por **x**, expresando explícitamente las consecuencias de dicha igualdad; si la igualdad es falsa, podemos suprimirla del antecendente. Formalmente realizamos reemplazamiento por igualdad entre el objetivo y sí mismo. El nuevo objetivo es:

	6. (z=x or z∈y::l) and x≤z and if w∈y::l then w≤z	z

La situación es similar a la anterior, con una disjunción entre dos proposiciones como primer conjuntado del objetivo. Esta vez tomamos partido por uno de los disjuntados (el primero), considerando que **z** y **x** son iguales. En el cuadro resolvemos el objetivo con el axioma **=-reflexiva**, en el que renombramos **x** como **x'**, utilizando el unificador {z<-x, x'<-x}:

	7. x≤x and if w∈y::l then w≤x	x

En este punto de la derivación hemos conseguido instanciar la variable de salida a un término básico. Por tanto, si conseguimos demostrar el objetivo, dicho término es un término de salida satisfactorio para la ecuación en derivación.

El primer conjuntado de nuestro objetivo es claramente cierto. Si resolvemos dicha proposición con el axioma **≤-refle-**

xiva, el nuevo objetivo es:

	8. if w∈y::l then w≤x	x

que se parte automáticamente en las filas:

9. w∈y::l		
	10. [w≤x]$^{+}$	x

El objetivo 10 contiene una condición que no podemos demostrar si es cierta o falsa, porque no podemos garantizar que un elemento **w** desconocido (pero relacionado con la lista **x::y::l**) es menor que **x**. Podemos intentar hacer un análisis de casos sobre la verdad y la falsedad de la condición. Esto significa que debemos abandonar esta rama de la deducción, comenzar otra que nos lleve a la condición contraria y entonces aplicar una regla (normalmente resolución) que derive un término condicional de salida. Sin embargo, antes de comenzar otra rama de deducción reescribimos el objetivo para que sea una condición ejecutable, es decir, primitiva.

El aserto 9 permite obtener una nueva conclusión mediante resolución con el aserto 2. Aplicando dicha regla con unificador {v<-w} se obtiene la fila:

11. [w≤maxlista y::l]$^{-}$		

Podemos ahora resolver por teoría las dos últimas filas, es decir, el objetivo 10 y el aserto 11, invocando la transitividad de la relación ≤:

if [x'≤y']$^{+}$ and y'≤z then [x'≤z]$^{-}$

con unificador {x'<-w, y'<-maxlista y::l, z<-x}. El objetivo

resultante es:

	10. [maxlista y::l≤x]$^{+}$	x

El nuevo objetivo es una condición primitiva, así que ya podemos abandonar temporalmente esta rama de la deducción.

Sean de nuevo el aserto 1 y el objetivo 6:

1. |maxlista y::l ∈ y::l|$^{-}$		
	6. (z=x or [z∈y::l]$^{+}$) and x≤z and if w∈y::l then w≤z	z

Recordemos que, en la deducción anterior tomamos partido por el primer disjuntado de la disjunción presente en el objetivo. Ahora vamos a considerar que el otro disjuntado es cierto. Formalmente, resolvemos el aserto 1 con el objetivo, tomando {z<-maxlista y::l} como unificador.

	11. x≤maxlista y::l and if w∈y::l then w≤maxlista y::l	maxlista y::l

El segundo conjuntado del objetivo es prácticamente igual que el aserto 2, así que podemos satisfacerlo resolviendo ambas filas, con unificador {v<-w}.

	12. [x≤maxlista y::l]$^{+}$	maxlista y::l

El nuevo objetivo es cierto si la cabeza del parámetro de la ecuación es menor o igual que el máximo encontrado en la cola del parámetro. Esta condición no puede satisfacerse a priori, pero es complementaria de la condición expresada por el objetivo 10. Por tanto, podemos realizar un análisis por

casos de las dos condiciones. De una manera formal en el sistema, resolvemos por teoría ambos objetivos invocando el axioma de totalidad de la relación ≤:

$$\boxed{x' \leq y'}^{-} \text{ or } \boxed{y' \leq x'}^{-}$$

con unificador {x'<-x, y'<-maxlista y::l}. El objetivo derivado es:

	13. true	if x≤maxlista y::l then maxlista y::l else x

El objetivo es **true**, así que podemos dar por concluida la demostración parcial. Sin embargo, aún realizamos una reescritura en el término de salida que, mediante la introducción de una definición local, aumente la eficiencia de la ecuación. El término definido localmente es la llamada recursiva **maxlista y::l**. La reescritura produce el objetivo final:

	13. true	let m == maxlista y::l in if x≤z then z else x

Ahora extraemos la ecuación resultante. Podríamos indicar que la ecuación se forme con una variable en la cabecera sinónima del subtérmino **y::l**. Se deja al gusto del programador decidir si merece la pena o no. Sin esta última sugerencia la ecuación derivada es:

```
--- maxlista x::y::l
      <= let m == maxlista y::l
         in if x≤m then m else x ;
```

La función completa derivada es:

```
dec maxlista : lista -> lista ;
--- maxlista x::nil
       <= x ;
--- maxlista x::y::l
       <= let m == maxlista y::l
          in if x≤m then m else x ;
```

4.4. ORDENACION DE LISTAS POR SELECCION

Un problema corriente de transformación de programas es la ordenación de listas (o de arrays). La razón estriba en la importancia de estos algoritmos para muchas aplicaciones. Por esto, la ordenación ha sido un problema sobre el que tradicionalmente se ha trabajado mucho, siendo [Knuth73] la referencia obligada. La investigación transformativa ha sistematizado aún más el estudio del problema de la ordenación. Además, se trata de un problema de una cierta dificultad, a diferencia de otros ejemplos "de juguete". En consecuencia ahora se conocen mucho mejor las decisiones de diseño que diferencian los distintos algoritmos de ordenación, pudiendo agruparlos en una familia de algoritmos [Merrit85]. Remitimos al lector interesado a otros artículos donde estos algoritmos se desarrollan sistemáticamente [Broy83, ClDa79, Darlington78, GrBa78, Smith85a, Traugott89].

Las derivaciones realizadas con nuestro sistema son similares a las detalladas (o simplemente comentadas) en [Traugott89]. Por tanto sólo vamos a derivar un algoritmo de ordenación por selección, que hemos elegido porque, junto con su función auxiliar, es un ejemplo de programa en cuya definición se usan extensivamente los patrones.

FUNCION PRINCIPAL

La especificación del problema es sencilla:

ordseleccion l:lista <= encontrar z:lista
tal que perm(l,z) and ord(z)

es decir, dada una lista **l**, queremos obtener una lista **z** que contenga los mismos elementos que **l** (es decir, que sea una permutación de **l**) pero que éstos estén ordenados (en orden ascendente no estricto).

La especificación deriva la declaración de tipo de **ordseleccion:**

dec ordseleccion : lista -> lista ;

Antes de comenzar la demostración de la especificación, incluimos los axiomas y propiedades que necesitaremos durante la derivación.

if $x \leq y$ and $y \leq z$ then $x \leq z$ ($\leq$-transitiva)

not x::l = nil (unicidad-nil)
nil<>l = l (<>-nil-izdo)
x::nil<>l = x::l (<>-átomo)
not $x \epsilon$ nil (ϵ-nil)
$x \epsilon$ y::l $\equiv$ x=y or $x \epsilon$ l (ϵ-::)
perm (l,l) (perm-reflexiva)
perm (x::l1,x::l2) $\equiv$ perm (l1,l2) (perm-::)
perm (x::l1,l2<>x::l3) $\equiv$ perm (l1,l2<>l3) (perm-<>)
ord nil (ord-nil)
if [ord l and if $w \epsilon$ l then $x \leq w$] then ord (x::l) (ord-::-sólosi)
if perm(l1<>x::l2,l3) then $l2 <_{saco} l3$ ($<_{saco}$)

El axioma **ord-::-sólosi** se ha obtenido tomando la rama

"sólo si" (pero no la "si") de la equivalencia del axioma **ord-::** (ver apartado B.3). Se necesita descomponer así la equivalencia para poder escolemizar la sentencia; en el caso "sólo si" se obtiene un término de Skolem **w(x,l)** que representamos con **w**.

También incluimos una propiedad a invocar en resolución por teoría:

$x \leq y$ or $y \leq x$ ($\leq$-totalidad)

La deducción la hacemos por inducción bien fundada y posterior descomposición en dos casos.

Lista vacía

El cuadro parcial recursivo correspondiente es:

asertos	objetivos	ordseleccion nil
1. if s<<nil then perm(s,ordseleccion s) and ord ordseleccion s		
	2. perm (nil,z) and ord z	z

La deducción de esta especificación es fácil, ya que la única lista que es una permutación de la lista vacía es ella misma. Además, la lista vacía está ordenada.

Formalmente, resolvemos el objetivo inicial con el axioma **perm-reflexiva**, con unificador {l<-nil, z<-nil}, y posteriormente resolvemos con el axioma **ord-nil**.

	3. ord nil	nil
	4. true	nil

La ecuación derivada es:

```
--- ordseleccion nil
      <= nil ;
```

Obsérvese que la demostración de esta especificación parcial no ha necesitado usar la hipótesis de inducción. La deducción para el caso vacío es igual en todas las derivaciones que usan descomposición; por tanto, es independiente de la estrategia de ordenación del algoritmo.

Caso general

El cuadro inicial recursivo es:

asertos	objetivos	ordseleccion x::l
1. if s<<x::l then perm (s,ordseleccion s) and ord ordseleccion s		
	2. perm (x::l,z) and ord z	z

donde **s**, **z** son variables y **x**, **l** son constantes de Skolem.

La idea básica de la ordenación por selección es extraer el elemento mínimo de la lista y unirlo al resto de la lista, ya ordenada. Por tanto, debemos partir la lista de salida en dos: el que será elemento mínimo y el resto.

Si redenominamos las variables **x**, **l**, del axioma **ord-::-sólosi** con los símbolos **z1**, **z2**, podemos resolver el axioma con el objetivo inicial. Usando el unificador {z<-z1::z2} nos queda:

	3. perm(x::l,z1::z2) and ord z2 and if w∈z2 then z1≤w	z1::z2

Obsérvese que el segundo conjuntado del nuevo objetivo

puede resolverse con el predicado **ord** contenido en la hipótesis de inducción. Instanciando **z** a **ordseleccion s**, la resolución de ambas sentencias produce el objetivo:

	4. perm (x::l,z1::ordseleccion s) and if w∈ordseleccion s then z1≤w and s<<x::l	z1::ordseleccion s

El objetivo puede simplificarse si utilizamos de nuevo la hipótesis de inducción, reemplazando **ordseleccion s** por **s**. Este reemplazamiento sólo es válido respecto a la relación **perm**, ya que si una lista **l1** es una permutación de otra lista **l2**, también son permutaciones **ordseleccion l1** y **l2**.

Aplicando reemplazamiento por permutación entre el objetivo 3 y la hipótesis de inducción obtenemos

	5. perm (x::l,z1::s) and if w∈s then z1≤w and s<<x::l	z1::ordseleccion s

Como parte de la demostración restante debemos encontrar una relación bien fundada << que sirva como base del principio de inducción. Dado que la lista **s** es la lista **x::l** sin el elemento mínimo **z1**, tomamos una relación que detecta la disminución de un número cualquiera de elementos de la lista: la relación $<_{\mathbf{saco}}$. Redenominamos la variable **x** del axioma $<_{\mathbf{saco}}$ con el símbolo **u**. Si ahora resolvemos el tercer conjuntado del objetivo con este axioma, tras aplicar el unificador {l2<-s, l3<-x::l}, obtenemos:

	6. perm (x::l,z1::s) and if w∈s then z1≤w and perm (l<>u::s,x::l)	z1::ordseleccion s

Los conjuntados primero y tercero del objetivo son prácti-

camente iguales. Tomemos **l** como **nil**. Reemplazando por igualdad el objetivo con el axioma **<>-nil-izdo**, se obtiene:

	7. perm(x::l,z1::s) and if w∈s then z1≤w and perm(u::s,x::l)	z1::ordseleccion s

Este último objetivo indica que el término **z1::ordseleccion(s)** es una salida satisfactoria si demostramos que la lista original **x::l** es una permutación de **z1::s** (y **u::s**) y que todos los elementos de **s** son mayores o iguales que **z1**.

La demostración directa de esta sentencia resulta difícil. Sin embargo, si demostramos el lema auxiliar (escolemizado):

```
[ if not l=nil
  then [perm(l,primero seleccionar l::segundo seleccionar l)
        and [ if v∈segundo seleccionar l
              then primero seleccionar l≤v ]                  ]]
```

la demostración de nuestro objetivo es inmediata. Este lema es la sentencia de correción de una función **seleccionar**. Por tanto, en este punto abandonaríamos temporalmente la demostración de **ordseleccion** para realizar la de **seleccionar**. Supongamos que hemos terminado satisfactoriamente ésta última; entonces podemos utilizar el lema anterior para concluir nuestra derivación.

Redenominemos la variable **l** de la sentencia de corrección de **seleccionar** con el símbolo **t**. Entonces resolvemos, usando unificación conmutativa, los conjuntados primero y tercero del objetivo con el primer conjuntado de la sentencia, siendo el unificador {t<-x::l, z1<-primero seleccionar (x::l), s<-segundo seleccionar (x::l), u<-primero seleccionar(x::l))}.

	8. [if wesegundo seleccionar (x::l))] [then primero seleccionar (x::l))≤w] and not x::l=nil	primero seleccionar (x::l) :: ordseleccion segundo seleccionar (x::l)

Análogamente podemos resolver el primer conjuntado del objetivo con el segundo conjuntado del consecuente de la sentencia de corrección de **seleccionar**, resultando un resolvente:

	9. not x::l=nil	primero seleccionar (x::l) :: ordseleccion segundo seleccionar (x::l)

Este objetivo es satisfecho por el axioma de unicidad del constructor nil, cuya resolución produce:

	10. true	primero seleccionar (x::l) :: ordseleccion segundo seleccionar (x::l)

La demostración ya ha terminado en este paso. Sin embargo el término de salida contiene funciones selectoras de pares, que deben ser eliminadas con una reescritura de definiciones locales. El nuevo objetivo es:

	11. true	let (m,r) == seleccionar (x::l) in m::ordseleccion r

De este modo la ecuación derivada es:

```
--- ordseleccion x::l
      <= let (m,r) == seleccionar x::l
         in m::ordseleccion r ;
```

También puede declararse, al crearse la ecuación, una variable sinónima del término **x::l**, lo que hace que **x**, **l** pasen a ser variables anónimas. La conveniencia de este paso se deja a la elección del lector.

FUNCION AUXILIAR DE SELECCION

La función **seleccionar** toma una lista no vacía y calcula el par formado por el menor elemento de la lista y el resto de la misma. Podemos especificarla con el formato opcional para funciones de rango tupla:

```
seleccionar  l:lista
          <=  encontrar  (z1,z2):entero#lista
              tal que  perm(l,z1::z2) and
                       (∀ y:entero) if y∈z2 then z1≤y
              donde not l=nil
```

La declaración de tipo derivada es:

```
dec seleccionar : lista -> entero#lista ;
```

Realizamos la demostración por inducción estructural sobre listas con tres casos.

Lista vacía

El cuadro inicial es el siguiente:

asertos	objetivos	seleccionar nil
1. not nil=nil		
	2. perm(nil,z1::z2) and if w∈z2 then z1≤w	(z1,z2)

Observamos que el cuadro es cierto de una manera inmediata, ya que contiene un aserto falso. Basta resolver el aserto 1 con el axioma **=-reflexiva** de listas y obtener:

3. false		

El nuevo aserto permite dar la deducción por terminada.

Puesto que no contiene término de salida, es un caso en el que no importa el valor de la función, así que no derivamos ecuación alguna.

Lista atómica

El segundo caso corresponde a una lista de entrada con un único elemento. El cuadro inicial es:

asertos	objetivos	seleccionar x::nil
	1. perm(x::nil,z1::z2) and if y∈z2 then z1≤y	(z1,z2)

La condición de entrada se ha suprimido automáticamente como aserto porque es una instancia del axioma de unicidad del constructor nil. Obsérvese que el término de salida recoge que la salida de la función es un par. En cuanto al objetivo, podemos comenzar la demostración notando que la única permutación de la lista atómica es ella misma. Por tanto, resolviendo el primer conjuntado con **perm-reflexiva**, con unificador más general {z1<-x, z2<-nil}, obtenemos:

	2. if y∈nil then x≤y	(x,nil)

Este objetivo es cierto de forma inmediata, pues no existe elemento **y** perteneciente a la lista vacía. Resolviendo el objetivo con el axioma **∈-nil** obtenemos:

	3. true	(x,nil)

El nuevo objetivo es evidentemente cierto, dando la deducción por concluida. La ecuación derivada es:

```
--- seleccionar x::nil
      <= (x,nil) ;
```

Lista general

Por último tenemos el caso en que la lista puede tener cualquier longitud mayor que uno, correspondiendo con el paso inductivo del principio de inducción estructural. El cuadro inicial, tras suprimir el aserto sobre la entrada (como en el caso anterior) y partir la hipótesis de inducción en dos asertos, es:

asertos	objetivos	seleccionar x::y::l
1. perm(y::l,primero seleccionar(y::l)::segundo seleccionar(y::l))		
2. if v∈segundo seleccionar(y::l) then primero seleccionar(y::l)≤v		
	3. perm(x::y::l,z1::z2) and if w∈z2 then z1≤w	(z1,z2)

La demostración de esta sentencia no es tan evidente como en los dos casos previos. Si el elemento menor de toda la lista de entrada fuera el primer elemento, **x**, la variable de salida **z1** debería tomar el valor **x**. Reflejamos este hecho resolviendo el primer conjuntado del objetivo con el axioma **perm-::**. Primero redenominamos la variable **x** del axioma con el nuevo símbolo **u** y posteriormente hacemos la resolución con unificador {u<-x, l1<-y::l, z1<-x, l2<-z2}.

	4. perm(y::l,z2) and if w∈z2 then x≤w	(x,z2)

Ya que hemos tomado la cabeza de la lista como su menor elemento, podemos tomar como resto de la lista la cola de la misma. Si resolvemos el objetivo, bajo unificador {z2<-y::l}, con el axioma **perm-reflexiva** resulta:

	5. if w∈ y::l then x≤w	(x,y::l)

El nuevo objetivo afirma que todos los elementos de la cola de la lista parámetro son menores que la cabeza de dicha lista. Es imposible demostrar esta afirmación ya que no hay ninguna restricción de tal estilo sobre el parámetro de entrada. Antes de abandonar esta rama de la deducción para buscar otra con un objetivo complementario, reexpresamos el objetivo de forma que sea una condición primitiva. Obsérvese que no hemos utilizado la hipótesis de inducción para derivar el término de salida, que por tanto no es recursivo; sin embargo sí lo necesitamos para reescribir el objetivo.

La pertenencia de un elemento a una lista es igualmente cierta si tomamos la lista o una permutación de la misma. Realizamos reemplazamiento por permutación entre nuestro objetivo y el aserto de inducción 1, resultando:

	6. if w∈primero seleccionar(y::l)::segundo seleccionar(y::l) then x≤w	(x,y::l)

El antecedente del objetivo expresa la pertenencia de un elemento a una lista con una cabeza y una cola. La misma subsentencia puede expresarse de forma equivalente con ayuda del axioma **∈-::**. Reemplazando por equivalencia entre el axioma y el objetivo queda:

	7. if w=primero seleccionar(y::l) or w∈segundo seleccionar(y::l) then x≤w	(x,y::l)

El antecedente del nuevo objetivo es una disjunción; si se cumple cualquiera de los dos conjuntados podemos afirmar el consecuente del objetivo. El objetivo puede reescribirse de una manera más natural mediante reemplazamiento por igualdad del objetivo consigo mismo, resultando:

	8. x≤primero seleccionar(y::l) and if [w∈segundo seleccionar(y::l)]$^-$ then [x≤w]$^+$	(x,y::l)

El primer conjuntado del objetivo nos da una condición primitiva, que la cabeza de la lista sea menor que el menor elemento de la cola de la lista. El segundo conjuntado afirma que la cabeza de la lista es menor que cualquier elemento de la cola que no sea el mínimo de la misma. Este conjuntado es deducible del aserto de inducción 2 y la propiedad de transitividad de la relación ≤, aunque la demostración es algo engorrosa, como vemos a continuación.

Resolvamos los antecedentes del segundo conjuntado del objetivo y del aserto 2:

if [v∈segundo seleccionar(y::l)]$^+$
then primero seleccionar(y::l)≤v

con unificador {v<-w}:

	9. x≤primero seleccionar(y::l) and not [primero seleccionar(y::l)≤w]$^-$	(x,y::l)

Podemos aplicar la propiedad transitiva de la relación ≤, previa redenominación de las variables **x**, **y**, **z** del axioma como **a**, **b**, **c**:

if a≤b and [b≤c]$^+$ then a≤c

Utilizando un unificador {b<-primero seleccionar(y::l), c<-w}, la resolución del objetivo y el axioma produce:

	10. x≤primero seleccionar(y::l) and not if a≤primero seleccionar(y::l) then [a≤w]$^-$	(x,y::l)

Si ahora resolvemos los consecuentes del segundo conjuntado de los objetivos 8 y 10, con unificador {a<-x}, conseguimos el objetivo primitivo buscado:

	11. x≤primero seleccionar(y::l)	(x,y::l)

Desarrollamos ahora otra rama de la demostración en la que el elemento menor de la lista no sea su propia cabeza. Recordemos el objetivo inicial 3:

	3. perm(x::y::l,z1::z2) and if w∈z2 then z1≤w	(z1,z2)

Si reemplazamos por igualdad en el conjuntado primero mediante el aserto **<>-átomo**, se obtiene el nuevo objetivo:

	12. perm(x::y::l,z1::nil<>z2) and if w∈z2 then z1≤w	(z1,z2)

Si redenominamos la variables **x**, **l2** del axioma **perm-<>** como **u**, **t**, podemos resolver el axioma y el objetivo, con unificador {u<-x, l1<-y::l, l2<-z1::nil, z2<-x::t}, quedando:

	13. perm(y::l,z1::nil<>t) and if w∈x::t then z1≤w	(z1,x::t)

Podemos deshacer el reemplazamiento que originó el objetivo 12, reemplazando por igualdad en el sentido contrario parte del segundo término de la relación **perm**.

	14. perm(y::l,z1::t) and if w∈x::t then z1≤w	(z1,x::t)

El antecedente del segundo conjuntado puede desdoblarse en una disjunción reemplazando por equivalencia el objetivo con el axioma ϵ-::.

	15. perm(y::l,z1::t) and if w=x or w∈t then z1≤w	(z1,x::t)

Si ahora reemplazamos por igualdad el objetivo consigo mismo, la implicación se formula como una conjunción más legible.

	16. [perm(y::l,z1::t)]$^+$ and z1≤x and if w∈t then z1≤w	(z1,x::t)

En este punto de la demostración tenemos dos conjuntados, el primero y el tercero, que pueden unificarse con los asertos de inducción. Si resolvemos el primer conjuntado con el aserto 1:

[perm
(y::l,primero seleccionar(y::l)::segundo seleccionar(y::l)]$^-$

con unificador {z1<-primero seleccionar(y::l), t<-segundo seleccionar(y::l)}, se consigue un objetivo con la forma:

	17. primero seleccionar(y::l)≤x and [if w∈segundo seleccionar(y::l) then primero seleccionar(y::l)≤w]$^+$	(primero seleccionar(y::l), x::segundo seleccionar(y::l))

Asimismo podemos resolver el aserto 2:

[if v∈segundo seleccionar(y::l)
then primero seleccionar(y::l)≤v]$^-$

con el segundo conjuntado, quedando:

	18. $\boxed{\text{primero seleccionar(y::l)}\leq x}^{+}$	(primero seleccionar(y::l), x::segundo seleccionar(y::l))

En este punto no podemos continuar por esta rama de la deducción porque no hay modo de asegurar que el menor elemento de la cola de la lista parámetro es menor que la cabeza de la misma. Sin embargo, el objetivo deducido es una sentencia complementaria del objetivo 11:

$$\boxed{x \leq \text{primero seleccionar(y::l)}}^{+}$$

Los dos objetivos no son resolubles directamente, sino que necesitamos relacionarlos previamente. Utilizando resolución por teoría, basta un solo paso de deducción para alcanzar el objetivo **true**. Tomando como propiedad invocada **≤-totalidad**:

$$\boxed{a \leq b}^{-} \text{ or } \boxed{b \leq a}^{-}$$

obtenemos:

	18. true	if x≤primero seleccionar(y::l) then (x,y::l) else (primero seleccionar(y::l), x::segundo seleccionar(y::l))

La derivación podría darse por concluida, pero podemos mejorar el término de salida suprimiendo las apariciones de funciones selectoras mediante una declaración local con patrones. El nuevo objetivo es:

	19. true	let (m,r) == seleccionar(y::l) in if x≤m then (x,y::l) else (m,x::r)

También puede introducirse durante la formación de la ecuación una variable sinónima del subtérmino **y::l**, pero no lo hacemos para no añadir más variables a la ecuación. La ecua-

ción derivada finalmente es:

```
--- seleccionar x::y::l
       <= let (m,r) == seleccionar y::l
          in if x≤m then (x,y::l) else (m,x::r) ;
```

Incluimos el programa derivado completo, formado por las funciones **ordseleccion** y **seleccionar:**

```
dec ordseleccion : lista -> lista ;
--- ordseleccion nil
       <= nil ;
--- ordseleccion x::l
       <= let (m,r) == seleccionar x::l
          in m::ordseleccion r ;

dec seleccionar : lista -> entero#lista ;
--- seleccionar x::nil
       <= (x,nil) ;
--- seleccionar x::y::l
       <= let (m,r) == seleccionar y::l
          in if x≤m then (x,y::l) else (m,x::r) ;
```

4.5. INTERSECCION DE DOS CONJUNTOS

Nuestro último ejemplo es la derivación de un programa que trata con un tipo de datos no predefinido en miniHope; además es un tipo de datos que no queremos que se comporte como un álgebra de palabras. Se trata del tipo de los conjuntos de enteros. Su declaración es:

```
data conjunto == ϕ ++ entero.conjunto ;
```

donde ϕ es un símbolo que representa el conjunto vacío y el punto '.' es el símbolo de función infija elegido para añadir un entero a un conjunto.

La teoría de conjuntos se aparta del esquema adaptado para las álgebras de palabras. No detallamos su definición aquí,

que ya se comentó en el apartado 2.4 y puede encontrarse en el apartado B.4.

Nuestro problema va a ser sintetizar una función que halle la intersección de dos conjuntos, es decir que dados dos conjuntos, calcule el menor conjunto cuyos elementos se encuentren en los dos conjuntos de entrada a la vez. Una especificación del problema es:

```
s:conjunto ∩ t:conjunto
          <=  encontrar  z:conjunto
              tal que  (∀ w:entero) w∈z ≡ w∈s and w∈t
```

donde representamos la intersección con el símbolo infijo usual ∩, de dominio el tipo de los pares de conjuntos de enteros. La especificación deriva la declaración de tipo de la función de intersección:

```
dec ∩ : conjunto#conjunto -> conjunto ;
```

Los dos únicos axiomas de la teoría que utilizamos en la derivación son:

```
not x∈φ                                              (∈-φ)
x∈y.s  ≡  x=y or x∈s                                 (∈-.)
```

Derivamos la función mediante una demostración por inducción estructural, sobre el conjunto **s**, con dos casos. Debemos señalar que el principio de inducción estructural normalmente adoptado para los conjuntos es ligeramente distinto de los usuales (ver el apartado 2.4 y el apartado B.4).

Conjunto vacío

El cuadro inicial es:

asertos	objetivos	$\phi \cap t$
	1. $w \epsilon z \equiv [w \epsilon \phi]^{\pm}$ and $w \epsilon t$	z

El objetivo expresa que la pertenencia de un elemento a la salida obliga a su pertenencia al conjunto vacío, pero esto es imposible, por lo que no puede haber entero que pertenezca a la salida.

Formalmente, comenzamos resolviendo el segundo predicado con el axioma ϵ-ϕ:

$$[\text{not } [x \epsilon \phi]^{+}]^{-}$$

deduciendo la fila:

	2. $[\text{not } w \epsilon z]^{+}$	z

Si resolvemos de nuevo con el axioma ϵ-ϕ, con unificador $\{z <- \phi\}$, obtenemos:

	3. true	ϕ

El objetivo deducido es el objetivo obvio, así que podemos formamos la ecuación derivada:

```
--- φ ∩ t
      <= φ ;
```

Conjunto no vacío

El caso de un conjunto no vacío es el paso inductivo de la demostración. Recordemos que este paso, digamos para el conjunto **x.s**, es ligeramente distinto del usual en listas porque contiene un antecedente que expresa que el entero **x** no perte-

nece al conjunto **s**. El cuadro inicial es:

asertos	objetivos	x.s ∩ t
1. not x∈s		
2. u∈snt ≡ u∈s and s∈t		
	3. w∈z ≡ w∈x.s and w∈t	z

Podemos desarrollar algo el segundo predicado de pertenencia reemplazando por equivalencia dicho predicado con el axioma **∈-.**, resultando:

	4. w∈z ≡ (w=x or w∈s) and w∈t	z

Podemos repetir el paso de deducción sobre el primer predicado del objetivo. Primero redenominamos las variables **x**, **s** del axioma con los símbolos **u**, **r**, y después reemplazamos por equivalencia, con unificador {u<-w, z<-y.r}, quedando el objetivo:

	5. w=y or w∈r ≡ (w=x or w∈s) and w∈t	y.r

Si logramos demostrar la verdad del objetivo, la salida estará formada por un elemento **y** añadido a un conjunto **r**. Ahora podemos reemplazar por igualdad el objetivo 5 sobre sí mismo, con unificador {y<-x} produciendo:

	6. x∈t and w∈r ≡ w∈s and w∈t	x.r

donde el elemento **y** del término de salida se ha instanciado a la constante de entrada **x**. El segundo conjuntado es prácticamente igual al segundo aserto de inducción. Resolviendo ambos tras unificación {u<-w, r<-s∩t}, resulta:

	7. $x \in t$	$x.(s \cap t)$

donde el término de salida es un término básico. El nuevo objetivo no puede demostrarse de momento, ya que no podemos afirmar si **x** pertenece o no a **t**. Por tanto, desarrollamos otra rama de la deducción que nos conduzca a la condición contraria, y así poder afirmar su disjunción.

Recordemos el objetivo 4:

	4. $w \in z \equiv (w=x \text{ or } w \in s) \text{ and } w \in t$	z

Esta vez no vamos a descomponer la salida **z** con el constructor punto. Realizamos reemplazamiento por igualdad del objetivo 4 consigo mismo.

	8. $x \in z \equiv x \in t$ and $w \in z \equiv w \in s$ and $w \in t$	z

El segundo conjuntado puede resolverse con el segundo aserto de inducción tras aplicar el unificador {u<-w, z<-s∩t}, deduciendo la fila:

	9. $x \in s \cap t \equiv x \in t$	$s \cap t$

obteniendo un término de salida básico. Si reemplazamos por equivalencia el lado izquierdo de la equivalencia mediante el segundo aserto de inducción, resulta:

	10. $\boxed{x \in s}^{\pm}$ and $x \in t \equiv x \in t$	$s \cap t$

Si ahora resolvemos el objetivo con el primer aserto de

inducción:

not $\boxed{x \epsilon s}^{+}$

se obtiene:

	11. not xεt	snt

El nuevo objetivo es igual que el objetivo 7 negado. Por tanto, su resolución nos conduce al objetivo inmediato:

	12. true	if xεt then x.(snt) else snt

La ecuación derivada es:

```
--- x.s ∩ t
       <= if xεt then x.(snt) else snt ;
```

En definitiva el programa derivado es:

```
dec ∩ : conjunto#conjunto -> conjunto ;
--- φ ∩ t
       <= t ;
--- x.s ∩ t
       <= if xεt then x.(snt) else snt ;
```

5. OTROS ASPECTOS

Existe un conjunto de aspectos y ampliaciones que no son parte esencial del sistema deductivo. Sin embargo, cada uno por una razón distinta, son interesantes de contemplar. Incluimos dichos aspectos en este capítulo, que tiene una estructura anárquica, de cajón de sastre. El primer aspecto tratado es el posible uso en el sistema deductivo de los principios de inducción estructural. Otra característica deseable del sistema es que pueda utilizarse, sin modificar la teoría combinada, para derivar programas en lenguajes con las mismas características pero sintaxis distintas (al menos con símbolos predefinidos distintos). Por último comentamos las características principales de un prototipo desarrollado de entorno interactivo de deducción.

5.1. INDUCCION ESTRUCTURAL EN EL SISTEMA DEDUCTIVO ATIPADO

Comparando los sistemas deductivos atipado y tipado, destaca su distinta filosofía. El primero obliga a tomar una decisión sobre la forma de demostrar una especificación, y por tanto de formar su cuadro inicial, recursivo o no recursivo. Como contrapartida los esquemas de demostración (y por tanto de programas) más corrientes no reciben ningún trato privilegiado. Por otro lado el sistema tipado obliga a tomar dos decisiones sobre el esquema de demostración, pudiendo resultar un cuadro recursivo (o no) y parcial (o completo). (La demostración por inducción estructural incorpora inducción y descomposición.) Esta mayor complejidad permite, sin embargo, realizar demostraciones más cortas y más intuitivas.

Dada la mayor versatilidad de nuestro sistema, podemos ver si alguna idea nuestra puede incorporarse al sistema atipado. Hay tres aspectos que pueden integrarse inmediatamente: la reescritura de definiciones locales, el esquema de especificación y demostración de funciones con rango tupla (apartado 3.7) y la garantía de transparencia consultiva de las funciones derivadas (apartado 3.9). Otro aspecto no tan evidente de incorporar es el uso de principios de inducción estructural distintos de la inducción bien fundada, es decir, principios de inducción estructural. En este apartado vemos que es posible su adición, pero a costa de una pérdida de claridad conceptual del sistema. Por concreción nos centramos en el uso del principio de inducción estructural sobre enteros. Pueden hacerse adaptaciones similares para demostraciones por descomposición o por otros principios de inducción.

Una función recursiva estructural expresada con una sola ecuación tiene un formato distinto que expresada con patrones. El esquema de ésta es:

--- f(0) <= t_1 ;
--- f(succ(n)) <= t_2[f(n)] ;

mientras que el de aquélla es:

f(n) <= if n=0 then t_1 else t_2[f(n-1)] ;

Las dos funciones son equivalentes, aunque el modo de recurrir es ligeramente distinto. En consecuencia, también el esquema de inducción implicado varía algo. En ambos casos utilizamos un principio de inducción estructural, sólo que en el caso de patrones se utiliza el principio por generación y en el otro caso, el principio por descomposición.

Recordemos que el principio de inducción estructural (sobre enteros) por descomposición se formula:

```
    ⌈ F[0]                                          ⌉
    |    and                                        |
if  |                                               |
    | (∀ x:num) ⌈ if not x=0                        ⌉|
    ⌊         ⌊ then if F[x-1]                      ⌋⌋
                      then F[x]
then (∀ x:num) F[x]
```

Podemos generalizar este principio para incorporar más casos básicos, quedando:

```
    ⌈ F[0]                                                          ⌉
    |    and                                                        |
    |   ...                                                         |
    |    and                                                        |
if  | F[k]                                                          |
    |    and                                                        |
    |                ⌈ if not x=0 and ... and not x=k              ⌉|
    | (∀ x:num)      ⌊ then ⌈ if F[x-1]                            ⌋|
    ⌊                       ⌊ then F[x]                             ⌋⌋
then (∀ x:num) F[x]
```

La validez de este principio se demuestra por inducción, de forma similar a los principios por generación del apartado 2.5).

La exposición que sigue se refiere por sencillez al principio por descomposición con dos casos, aunque puede verse que su adaptación al caso genérico es inmediata. También limitamos el estudio, sin pérdida de generalidad, a un solo parámetro (entero). Comenzamos reformulando las propiedades y proposiciones que identifican el cuadro inicial recursivo y garantizan su corrección.

Propiedad (término de salida estructural, enteros)

En una teoría de enteros, si los términos básicos $\bar{t}[a]$ satisfacen el cuadro:

	Q[0;$\bar{z}_1$] and if num(a) then if not a=0 then if Q[a-1;$\bar{f}$(a-1)] then Q[a;$\bar{z}_2$]	if a=0 then $\bar{z}_1$ else $\bar{z}_2$

donde **a** es una constante nueva (de Skolem) y $\bar{z}_1$, $\bar{z}_2$ son las únicas variables libres en **Q[0;**$\bar{z}_1$**]** y **Q[a;**$\bar{z}_2$**]**, respectivamente, entonces la sentencia

```
Q[0;t̄[0]]
   and
(∀ x:num) [ if not x=0
            then if Q[x-1;f̄(x-1)]
                 then Q[x;t̄[x]] ]
```

es válida.

Demostración:

Supongamos que los términos $\bar{t}$**[a]** satisfacen el cuadro anterior. Entonces debemos mostrar que la sentencia

```
Q[0;t̄[0]]
   and
(∀ x:num) [ if not x=0
            then if Q[x-1;f̄(x-1)]
                 then Q[x;t̄[x]] ]
```

es válida.

Por la regla de eliminación de cuantificador universal y expandiendo los cuantificadores relativizados basta demostrar que la sentencia

```
Q[0;t̄[0]]
   and
if num(a)
then if not a=0
     then if Q[a-1;f̄(a-1)]
          then Q[a;t̄[a]]
```

donde **a** son constantes nuevas, es cierta bajo cualquier modelo **I**.

Dado que los términos $\bar{t}$**[a]** satisfacen el cuadro, para alguna sustitución acomodadora **θ** se cumple la condición de verdad, es decir, la sentencia

$$\left[\begin{array}{l} Q[0;\bar{z}_1] \\ \quad \text{and} \\ \text{if num(a)} \\ \text{then if not a=0} \\ \qquad \text{then if } Q[a-1;\bar{f}(a-1)] \\ \qquad\quad \text{then } Q[a;\bar{z}_2] \end{array}\right]\theta$$ es cerrada y cierta bajo I

y la condición de salida, es decir, los términos

(if a=0 then $\bar{z}_1$ else $\bar{z}_2$)θ son básicos y tienen el mismo valor que $\bar{t}$[a] bajo I.

o, de forma equivalente, ya que **a** es una constante,

if a=0 then $\bar{z}_1\theta$ else $\bar{z}_2\theta$ son básicos y tienen el mismo valor que $\bar{t}$[a] bajo I.

Esto significa (por la semántica del término if-then-else y la propiedad de sustitutividad de la igualdad) que si **a=0** es cierto bajo **I**,

$\bar{z}_1\theta$ tienen el mismo valor que $\bar{t}$[0]

y si **a=0** es falso bajo **I**,

$\bar{z}_2\theta$ tienen el mismo valor que $\bar{t}$[a]
(para un **a** con valor distinto de cero).

Dado que las únicas variables libres de la condición de verdad son $\bar{z}_1$, $\bar{z}_2$, esta condición es equivalente a afirmar que

```
Q[0;z̄1θ]
   and
if num(a)
then if not a=0
     then if Q[a-1;f̄(a-1)]
          then Q[a;z̄2θ]
```

es cerrada y cierta bajo **I**, o equivalentemente, por las consecuencias de la condición de entrada, que

```
Q[0;t̄[0]]
   and
if num(a)
then if not a=0
     then if Q[a-1;f̄(a-1)]
          then Q[a;t̄[a]]
```

es cierta bajo I. Pero ésta es la sentencia que queríamos demostrar. ∎

Definición (cuadro inicial recursivo estructural, enteros)

Dada una especificación **Q[x;z̄]**, para **x** de género entero, el cuadro inicial para su demostración por inducción estructural (con dos casos) tiene la forma:

asertos	objetivos	$\bar{f}(a)$
	$Q[0;\bar{z}_1]$ and if num(a) then if not a=0 then if $Q[a-1;\bar{f}(a-1)]$ then $Q[a;\bar{z}_2]$	if a=0 then $\bar{z}_1$ else $\bar{z}_2$

donde **a** son constantes nuevas. ∎

Proposición (cuadro recursivo inicial estructural, enteros)

En cualquier teoría,

si los términos cerrados $\bar{t}$**[a]** satisfacen el cuadro inicial recursivo estructural de una especificación,

entonces el programa $\bar{f}$**(x)=**$\bar{t}$**[x]** satisface la especificación.

Demostración:

Para demostrar que el programa $\bar{f}(x)=\bar{t}[x]$ satisface la especificación, debe demostrarse la validez de la condición de corrección:

$$\text{if } (\forall\ x{:}\text{num})\ \bar{f}(x)=\bar{t}[x]$$
$$\text{then } (\forall x{:}\text{num})\ Q[x;\bar{f}(x)]$$

en la teoría.

Supongamos que bajo cierto modelo de la teoría

$$(\forall\ x{:}\text{num})\ \bar{f}(x)=\bar{t}[x]$$

es cierto, y queremos demostrar que

$$(\forall\ x{:}\text{num})\ Q[x;\bar{f}(x)]$$

también es cierto.

Dado el principio de inducción estructural para enteros, basta demostrar la validez de:

$$Q[0;\bar{f}(0)]$$
$$\text{and}$$
$$(\forall\ x{:}\text{num})\ \left[\begin{array}{l}\text{if not } x=0\\ \text{then if } Q[x-1;\bar{f}(x-1)]\\ \qquad\text{then } Q[x;\bar{f}(x)]\end{array}\right]$$

o, según la asunción de corrección, la validez de:

$$Q[0;\bar{t}[0]]$$
$$\text{and}$$
$$(\forall\ x{:}\text{num})\ \left[\begin{array}{l}\text{if not } x=0\\ \text{then if } Q[x-1;\bar{f}(x-1)]\\ \qquad\text{then } Q[x;\bar{t}[x]]\end{array}\right]$$

Según la propiedad del término estructural de salida (para enteros), para demostrar que la sentencia anterior es válida, basta encontrar términos básicos $\bar{t}[a]$ que satisfagan el cuadro

inicial. Pero hemos supuesto que esto ocurre. ∎

Resumiendo, es posible adaptar el cuadro deductivo atipado para la derivación directa de programas estructurales. El cuadro recursivo inicial es intuitivo y por tanto fácil de usar. Sin embargo, debe demostrarse una propiedad adicional del término de salida. La adición de nuevos esquemas privilegiados de demostración provocaría una profusión de versiones de dicha propiedad que puede resultar algo confuso. Este inconveniente no aparece en el cuadro tipado porque existe una única propiedad del término de salida parcial.

5.2. TEORIA LOGICA INDEPENDIENTE DEL LENGUAJE DE PROGRAMACION

Una característica del sistema deductivo es que los símbolos de tipo, función y predicado de las teorías lógicas y de los programas funcionales coinciden. Por ejemplo, la función de concatenación de listas se representa en ambos lugares con el símbolo <>. En consecuencia el sistema resulta uniforme y sencillo. Sin embargo, el sistema así definido es algo rígido, puesto que si queremos cambiar miniHope por otro lenguaje de programación, también debemos reescribir la teoría lógica combinada con los nuevos símbolos.

La consecución de un sistema deductivo donde la teoría lógica sea independiente del lenguaje de programación es sencilla. Basta observar que la introducción de los símbolos de función o predicado en los términos de salida se produce por instanciación de las variables lógicas. Podemos controlar este proceso de instanciación para introducir los símbolos propios del lenguaje de programación en cuestión. Veamos el mecanismo exacto.

Supongamos que disponemos de una teoría lógica con símbolos cualesquiera. Por ejemplo, podemos elegir ≤, =<, **lesseq** o p_{44} para representar la relación "menor o igual" entre enteros. Sea

$$S_T = \{f_T, g_T, h_T, \ldots\}$$

el conjunto de símbolos de la teoría **T**. Queremos derivar programas en cierto lenguaje funcional **L** con patrones. El lenguaje consta de un conjunto S_L de símbolos de función, predicado y tipo predefinidos.

Definición (función de transformación de símbolos)

Definimos la función **TS-TL** (o simplemente **TS-L**) como una función que transforma símbolos de S_T en símbolos de S_L. (**TS-TL** significa Transformación de Símbolos de la teoría T al lenguaje L; si la teoría T siempre es la misma, se sobreentiende cuál es su nombre.)

En general una función **TS-L** es una función parcial, ya que sólo está definida para los símbolos de S_T que corresponden a símbolos predefinidos de S_L. Ampliamos la función hasta hacerla total, asociando a cada símbolo de S_T sin imagen en S_L un símbolo nuevo cualquiera, correcto según la sintaxis del lenguaje **L** (p.ej. él mismo si es correcto en **L**). ∎

Ejemplo (TS-miniHope)

Si definimos la función **TS-miniHope**, a los símbolos + y ≤ de la teoría de los enteros les asociamos los símbolos predefinidos respectivos + y =<. Asimismo al símbolo ^{2} le asociamos el símbolo no predefinido **cuadrado**; no podemos asociarle él mismo porque es incorrecto utilizar caracteres no alfabéticos como identificador de una función no infija. Por último supongamos que los tipos predefinidos en miniHope tuvieran el mismo

símbolo que en Hope; el símbolo miniHope **num** se asociaría al símbolo **entero** de la teoría. ∎

Con ayuda de la función **TS-L** desarrollamos una función que transforma expresiones de la teoría en términos del lenguaje.

Definición (función de transformación de términos)

Sea una función de transformación de símbolos **TS-L**. Definimos la función **TT-L** (asociada a **TS-L**) de transformación de expresiones de la teoría en términos del lenguaje **L** de la manera siguiente, por casos sobre la estructura del término:

- Una constante **c**:
  TT-L(c) = TS-L(c)
- Una variable **v**:
  TT-L(v) = v
- Una aplicación de una función **f** sobre términos $t_1,\ldots,t_n$:
  TT-L(f($t_1,\ldots,t_n$))
  = (TS-L(f)) (TT-L(t_1),...,TT-L(t_n))
- Una proposición de predicado **p** y términos $t_1,\ldots,t_n$:
  TT-L(p($t_1,\ldots,t_n$))
  = (TS-L(p)) (TT-L(t_1),...,TT-L(t_n)) ∎

Ejemplo (TT-miniHope)

Sea la expresión

$$(z+i)^2 \leq n$$

La función TT-miniHope asociada a TS-miniHope transforma esta expresión en:

cuadrado(z+i)<=n ∎

Definición (sustitución con transformación)

Sea una sustitución $\theta = \{x_1<-t_1,\ldots,x_n<-t_n\}$ y una función **TS-L** de transformación de símbolos. La sustitución con transformación θ_{TS-L} se define como una sustitución igual a θ, excepto que se transforman, mediante la función **TT-L** de transformación de términos asociada a **TS-L**, todos los términos t_i de θ. Es decir:

$$\theta_{TT-L} = \{x_1<-TT\text{-}L(t_1),\ldots,x_n<-TT\text{-}L(t_n)\}$$ ∎

Ejemplo ($\theta_{TS-miniHope}$)

Dada la sustitución $\theta=\{z<-z'+i\}$, la nueva sustitución $\theta_{TS-miniHope}$ tiene el mismo valor $\{z<-z'+i\}$. ∎

Ahora podemos modificar ligeramente la definición de las reglas de deducción para hacerlas independientes del lenguaje de programación. Las modificaciones simplemente hacen que la instanciación de las variables de salida y la introducción de predicados sólo produzcan símbolos de S_L. Básicamente, para cada lenguaje **L** y transformación de símbolos **TS-L** modificamos las reglas de modo que las instanciaciones θ sobre los términos de salida sean θ_{TS-L} y los predicados se traduzcan con **TT-L**. Sólo necesitan modificarse aquellas reglas de deducción que introducen sustituciones en el término de salida.

Veamos como ejemplo la nueva definición de la forma básica de la regla de resolución. Para el resto de las reglas la modificación es análoga.

Regla de resolución (independiente del lenguaje)

Dados dos asertos $\mathbf{A_1}$ y $\mathbf{A_2}$ estandarizados mutuamente:

$A_1[P]$		s
$A_2[P']$		t
$A_1\theta$[false] and $A_2\theta$[true]		if TT-L(P)θ_{TS-L} then sθ_{TS-L} else tθ_{TS-L}

donde las restricciones sobre el cuadro son las mismas que en la formulación original de la regla de resolución (apartado 3.4). ∎

Ejemplo (regla de resolución independiente del lenguaje)

Sea el objetivo inicial de la derivación de una función **raizcuad** (comparar con ejemplo en el apartado 3.4):

	$\boxed{z^2 \leq n}^+$ and not $(z+i)^2 \leq n$	z

y una copia del mismo con las variables redenominadas:

	$z^2 \leq n$ and not $\boxed{(z+i)^2 \leq n}^-$	z

Aplicando la regla (nueva) de resolución a ambos objetivos obtenemos el nuevo objetivo:

	$z'^2 \leq n$ and not $(z'+i+i)^2 \leq n$	if cuadrado(z'+i)=<n then z'+i else z'

donde el término de salida está expresado en miniHope. ∎

Observación (declaración de tipos)

La independencia del lenguaje debe extenderse también a la declaración de tipos de las funciones. Por tanto la declaración de tipos se construye de la manera usual utilizando **TS-L(D)** y **TS-L(R)** como símbolos respectivos del tipo del dominio y del rango, en vez de los usuales **D** y **R**. De esta manera, para la función **raizcuad**, cuya especificación contiene:

raizcuad (n,i):entero#entero <= encontrar z:entero
...

se produce la declaración de tipos:

dec raizcuad : num#num -> num ; ∎

Observación (independencia de la sintaxis ecuacional)

Los mecanismos presentados mantienen la independencia de la teoría respecto a los símbolos del lenguaje de programación. Sin embargo, la sintaxis de las declaraciones de tipos y de las ecuaciones permanece invariada, así que debe modificarse el método de formación de una ecuación a partir del cuadro final. En algunos lenguajes las ecuaciones de una función no se escriben independientemente unas de otras, por lo que también se necesita un mecanismo que relacione las ecuaciones. ∎

5.3. ENTORNO INTERACTIVO DE DERIVACION

Como se comentó en el apartado 1.1, un fin último de los sistemas transformativos es la automatización de la producción

de software, aunque su automatización completa es una tarea imposible hoy en día [RiWa88]. En un sistema deductivo se precisa la demostración automática de teoremas. Aunque se ha avanzado mucho en este campo, el objetivo final todavía está lejano [Wos85, Wos87].

Una opción menos ambiciosa es construir un entorno interactivo donde el usuario controla totalmente el sentido de la deducción, pero el entorno realiza los detalles rutinarios de la misma. Un simple entorno interactivo proporciona dos ventajas. Primero, la realización por el entorno de los aspectos rutinarios permite que el usuario derive programas más rápidamente que de forma manual y sin cometer fallos durante la derivación. Pero también facilita que se puedan conocer mejor las ventajas e inconvenientes del método desde el punto de vista del usuario. Cuanto más conocimiento se posea, más tareas podrá incorporar el entorno, que así irá evolucionando hacia un entorno semiautomático.

A continuación describimos las características funcionales y la interfaz de usuario de un hipotético entorno que incorpore nuestro método deductivo. Una buena parte de sus características se inspiran en un prototipo de entorno [Mañas90] desarrollado que incorpora el primer método deductivo de Manna y Waldinger [MaWa80].

CARACTERISTICAS FUNCIONALES

(En lo sucesivo utilizamos el término función para indicar una tarea que puede hacer el entorno; las funciones o programas funcionales los llamamos simplemente programas.)

El entorno contiene una serie de funciones que en conjunto permite derivar deductivamente programas. Básicamente se trata de incluir las funciones necesarias para realizar el proceso deductivo (similar al descrito en el apartado 3.9) junto con

la flexibilidad necesaria para simultanear otras operaciones o deshacer errores cometidos. El número de funciones es mediano, así que resulta conveniente agruparlas en cuatro grupos, según su característica principal. Además puede incluirse una función adicional de introducción al método y al entorno. Veamos con más detalle los otros cuatro.

Funciones de especificación

Denominamos así a las funciones que preparan el proceso deductivo propio. Distinguimos cuatro:

- Edición de la teoría lógica. Permite añadir, modificar y borrar axiomas de la teoría lógica combinada. Los axiomas se encuentran clasificados, al menos por el tipo de los datos implicados (ver como ejemplo la presentación de la teoría en el Apéndice B). La estructura resultante de almacenamiento de los axiomas puede ser arborescente (similar a la organización de clases y métodos en un entorno Smalltalk-80).
- Especificación de funciones. Se presenta al usuario un armazón de especificación endulzada que debe rellenar con los símbolos de las funciones a derivar, variables de entrada y de salida y condiciones de entrada y de salida.
- Elección del esquema de demostración. Debe elegirse un esquema no inductivo, inductivo bien fundado o inductivo estructural. Los dos primeros llevan a considerar un esquema de descomposición. En los casos inductivos debe decidirse cuáles son los parámetros de inducción. En los casos inductivo estructural y descompositivo debe decidirse el parámetro a descomponer y el número de casos básicos.
- Identificación de funciones no primitivas. El entorno incorpora esta lista de funciones para evitar su introducción en los términos de salida.

Funciones deductivas

Estas funciones realizan operaciones de demostración de una especificación:

- Selección de cuadro parcial. Se elige el cuadro parcial con el que se quiere trabajar. Si antes se estaba trabajando con otro cuadro, se conserva la demostración parcial realizada.
- Aplicación de regla de deducción. Debe identificarse la regla a aplicar y las filas y subsentencias implicadas. Las reglas de reemplazamiento y unificación por relación son difíciles de implementar ya que dependen de cada relación binaria concreta. Una decisión inicial puede ser incorporar solamente las reglas de reemplazamiento por igualdad y por equivalencia.
- Deshacer pasos de deducción. Deshace cierto número de pasos realizados previamente. No es necesario que los pasos a deshacer sean los últimos del cuadro, pero sí que sean los últimos de una rama de deducción.

Funciones de consulta

Son funciones que permiten examinar, sin poder modificarlas, sentencias no visibles en ese momento en la pantalla. Es decir, es un examinador ("browser"), no un editor.

- Consulta de la teoría lógica.
- Consulta de las filas anteriores.

Funciones de ficheros

Junto a las funciones propias del método deductivo, existe un grupo de funciones de comunicación con el mundo exterior que todo entorno interactivo debe incluir. Las hemos denominado funciones de ficheros porque interactúan principalmente con ficheros en disco.

- Recuperar y almacenar deducciones. Permiten leer y escribir deducciones previas, terminadas o no.
- Imprimir. Escribe por impresora la deducción en curso realizada hasta el momento, facilitando así que el programador analice la sesión transcurrida. En esta función y en las dos anteriores no sólo está involucrado el cuadro parcial actual, sino la especificación el esquema de demostración, la lista de funciones no primitivas y otros cuadros parciales.
- Directorio. Da la lista de ficheros agrupados bajo cierto directorio.
- Salida temporal al sistema operativo. La deducción en curso se conserva en memoria.
- Salida definitiva al sistema operativo.

Mecanismos rutinarios automatizados

El entorno realiza automáticamente un conjunto de labores rutinarias, facilitando así la labor del programador. Algunas de ellas son propensas a la comisión de errores, otras son simplemente repetitivas. Destacamos:

- Cálculo de polaridad de subsentencias y comprobación de la correcta aplicación de las reglas de resolución y reemplazamiento.
- Unificación y sustitución, como parte de la aplicación de una resolución o un reemplazamiento. La unificación se hace sobre sentencias estandarizadas mutuamente, renombrándose variables si es necesario.
- Aplicación automática de las reglas de simplificación y partición. Tras crear un cuadro o aplicar una regla de deducción, el cuadro se simplifica y se parte lo máxino posible.
- Comprobación de fin de deducción parcial tras cada paso de deducción. En caso afirmativo se avisa al usuario.
- Registro de si la deducción se ha almacenado. Esto hace

que en caso de salir definitivamente al sistema operativo, introducir una nueva especificación o elegir un nuevo esquema de demostración, se avise primero al programador de que quizá deba salvar la deducción previa.

INTERFAZ DE USUARIO

La presentación de información al usuario es una tarea delicada, a veces más difícil que la identificación de las funciones que debe incorporar el entorno. Se ha optado por un sistema de menús ayudado con un sistema de ventanas. En el nivel superior del entorno hay cuatro opciones correspondientes a los grupos de funciones disponibles. Una opción puede elegirse con un ratón o pulsando una tecla asociada con la opción, y provoca la aparición de un submenú con nuevas opciones. A su vez algunas de estas opciones pueden hacer aparecer otros menús o pedir algún dato al usuario.

Como ejemplo, la figura muestra el menú principal más el submenú de funciones sobre ficheros. Cada letra en negrita representa la tecla a pulsar para elegir la opción correspondiente.

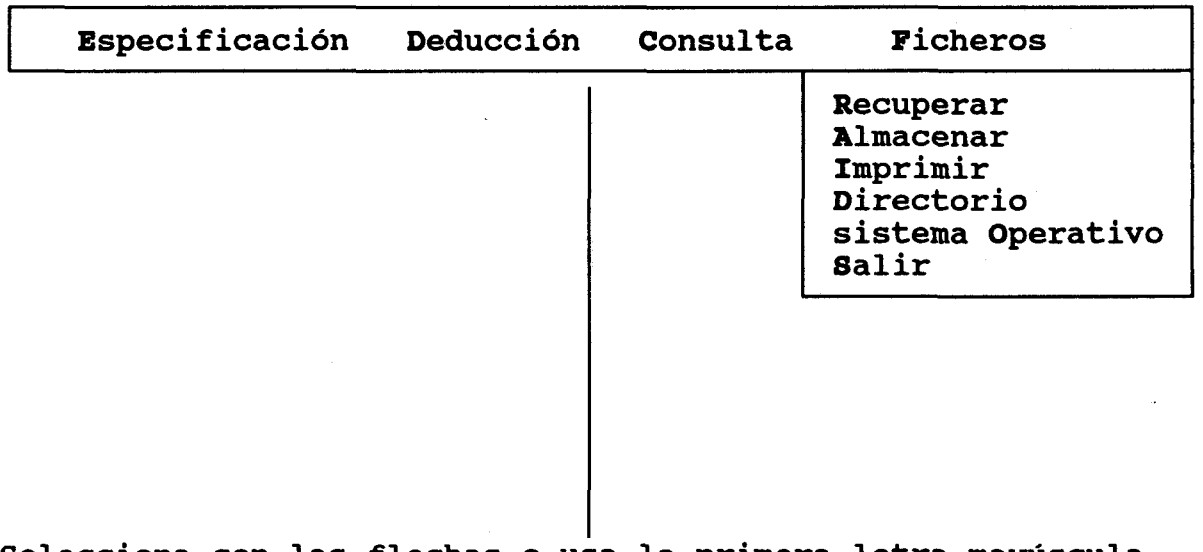


Figura 5.1. Interfaz de menús del entorno interactivo

El hardware impone las principales restricciones sobre la organización de la interfaz de usuario. Por ejemplo, las pantallas convencionales para PC tienen poca resolución, lo que hace que limita la cantidad de información visualizable. Una opción es mostrar únicamente dos columnas, como aparece en la figura anterior; la primera columna contiene asertos y objetivos y la segunda términos de salida. Cada sentencia se encuentra prefijada por una letra indicadora de aserto u objetivo ("A" u "O" respectivamente) y un número de fila.

6. CONCLUSIONES

Recapitulando sobre el trabajo realizado, vemos que permite obtener programas funcionales con patrones de una manera formal. Se parte de una sentencia lógica, que constituye una especificación del programa deseado, y se demuestra su validez en una teoría lógica, generando como producto secundario un programa funcional que satisface la especificación. El uso de la lógica en el proceso garantiza la corrección del programa obtenido. El método se basa en el cuadro deductivo de Manna y Waldinger [MaWa80, MaWa87] y forma parte de la tradición deductiva de la programación mediante transformaciones.

El desarrollo del método deductivo se ha efectuado en dos etapas. En primer lugar se han desarrollado los principios lógicos necesarios para la derivación de funciones con patrones, pero sin comprometerse con ningún sistema formal. Entre los aspectos lógicos tratados, se ha dedicado cierta atención a la caracterización de las teorías lógicas de aquellos tipos de datos que son álgebras libres. También se han identificado varios esquemas de demostración de sentencias mediante la demostración de subsentencias independientes. Se han estudiado dos modos de obtener subespecificaciones, por descomposición y por inducción estructural.

Las ideas anteriores se han adaptado a un sistema formal de derivación de programas: el cuadro deductivo. La derivación de funciones con patrones implica la descomposición de la especificación completa en especificaciones parciales. Estas se demuestran en cuadros parciales distintos, produciendo cada uno una ecuación funcional. La descomposición de la especificación se hace gracias a un razonamiento por descomposición o por inducción estructural. En el primer caso también puede

usarse inducción bien fundada, mientras que el segundo ya incorpora una forma de inducción. La identificación de los esquemas de demostración necesarios, junto con su adaptación al cuadro forman el logro más importante de la tesis.

Podemos mencionar algunos otros aspectos del método deductivo desarrollado. Primero, se ha identificado una reducción en la derivación de algunas funciones con rango tupla. Segundo, las ecuaciones derivadas contienen definiciones locales con patrones para evitar el uso de funciones selectoras. Tercero, se contempla la derivación de ecuaciones con variables anónimas y sinónimas en su cabecera. Cuarto, se incluye un proceso completo de derivación, a modo de manual de usuario. Quinto, se ha desarrollado un criterio para que las funciones auxiliares derivadas no contengan variables globales.

El método desarrollado se ha aplicado a una gama de ejemplos. Podemos destacar la derivación de una familia de cuatro funciones distintas de ordenación de listas; hemos incluido la derivación completa del algoritmo de ordenación por selección.

Como ampliaciones del método deductivo, se ha modificado ligeramente para que el vocabulario de la teoría lógica sea independiente del vocabulario del lenguaje de programación. Asimismo hemos identificado algunas características de un entorno interactivo de derivación, especialmente las funcionalidades principales del entorno y la interfaz de usuario.

El resultado final de la tesis es un sistema de síntesis de funciones con patrones, que es general gracias a la lógica usada. El sistema anterior de Manna y Waldinger tiene una capacidad similar, pero orientado a lenguajes funcionales antiguos, es decir, lenguajes atipados como Lisp. Para lenguajes tipo Hope sólo existen, que sepa el autor, sistemas con una capacidad limitada de síntesis, como es el sistema de despliegue-pliegue de Burstall y Darlington. En consecuencia el autor espera haber contribuido al área de la programación

transformativa con un método deductivo de obtención de programas funcionales con patrones.

La tesis puede ser continuada de varias maneras, destacando cuatro grupos de actividades: ampliación de las clases de programas derivables, desarrollo de entornos interactivos más elaborados, estudio de sus relaciones con otros métodos transformativos y estudio de tácticas y estrategias de automatización.

Hay gran número de funciones no derivables debido a limitaciones de la lógica multigénero utilizada. Por ejemplo, no podemos derivar funciones definidas sobre tipos de datos polimórficos porque nuestra lógica es monomórfica. Se precisa utilizar una lógica polimórfica con géneros disjuntos. Tampoco podemos derivar predicados. La lógica debería hacer coincidir los conceptos de función y predicado para que la columna de salida pueda recoger tanto términos como sentencias. Un tercer grupo de funciones no derivables son las funciones de orden superior. Aparte de necesitar una lógica de orden superior, hay detalles no triviales que considerar, como la unificación.

Resulta natural la extensión del método a la derivación de programas en otros estilos declarativos. Algunos no parecen presentar problemas, como la derivación de ciertas ecuaciones condicionales. Un campo relacionado es la derivación de programas lógicos [ClSi77, HaTä79, Hogger81, TaSa84], pero no se ha estudiado su realización con el cuadro deductivo. Este trabajo conduciría de manera natural a la derivación de programas en lenguajes lógico-funcionales [DeLi86, Moreno89].

Otra línea de investigación es el desarrollo de un entorno deductivo más elaborado. Los aspectos técnicos del entorno no tienen una respuesta inmediata. Nuestra experiencia es que hay aspectos nuevos, pero que la mayor parte son comunes con entornos convencionales, por lo que pueden ser fácilmente

adaptados. Por ejemplo, un buen número de características pueden modelarse como en los entornos orientados a estructuras [ReTe87].

Una facilidad interesante en un entorno transformativo es la disponibilidad de un lenguaje de registro [Wile83, Reddy88]. Además de documentar cada derivación, permitiría reutilizar partes de derivaciones previas. También sería interesante que la estructura del sistema fuera flexible para experimentar con tácticas y estrategias de deducción automática o semiautomática.

Una tercera línea de trabajo, relacionada con la primera, es estudiar la relación del método deductivo con otros. Además de su interés puramente teórico, un mejor conocimiento de estas relaciones puede facilitar la integración de distintos sistemas o la simulación de un sistema por otro. Por ejemplo, el método de Burstall y Darlington puede concebirse como una forma particular de realizar reemplazamientos por igualdad. Esto también permite incorporar al sistema o a una restricción del mismo heurísticas que se han mostrado afortunadas en otros sistemas [Darlington81a, Reddy88, Smith85b]. En este punto se enlaza con la siguiente línea de trabajo.

El último campo de trabajo previsible es el estudio de tácticas y estrategias para la semiautomatización del proceso deductivo. Esta área no sólo interesa a la programación transformativa, sino a la demostración de teoremas [Aubin79, BoMo75] y otros campos de la inteligencia artificial relacionados con búsqueda y control. Pueden desarrollarse estrategias (quizá no completas) de búsqueda y compaginarse con un uso manual del sistema completo.

BIBLIOGRAFIA

En las referencias siguientes se utilizan las siguientes abreviaturas de revistas y congresos:

AI - Artificial Intelligence.
CACM - Communications of the ACM.
CJ - The Computer Journal
IJCAI - International Joint Conference on Artificial Intelligence.
JACM - Journal of the ACM.
JAR - Journal of Automated Reasoning.
JLP - Journal of Logic Programming.
JSC - Journal of Symbolic Computation.
SCP - Science of Computer Programming.
TCS - Theoretical Computer Science.
TOPLAS - ACM Transactions on Programming Languages and Systems.

También se incluyen las siguientes abreviaturas de términos:

Conf. - Conference.
Eng. - Engineering.
Int. - International.
Lang. - Languages.
Princ. - Principles.
Proc. - Proceedings.
Prog. - Programming.
Soft. - Software.
Symp. - Symposium.
Trans. - Transactions.

La relación bibliográfica es la siguiente:

[AbSuSu85] H. Abelson, G.J. Sussman y J. Sussman, Structure and Interpretation of Computer Programs, The MIT Press, 1985.

[ACM90] "Scaling-up: A research agenda for software engineering", Report of the Computer Science and Technology Board. Versión reducida en CACM, 33(3):281-293, marzo 90.

[Agresti86] W.W. Agresti, New Paradigms for Software Development, IEEE Computer Society Press, 1986.

[Aubin79] R. Aubin, "Mechanizing structural induction", TCS, 9:329-362, 1979.

[BaChGr83] R. Balzer, T.E. Cheatham y C. Green, "Software technology in the 1990's: Using a new paradigm", Computer, 16(11):39-45, noviembre 83. También en [Agresti86], págs.16-22.

[BaFe82] A. Barr y E.A. Feigenbaum (dir.), The Handbook of Artificial Intelligence, vol.II, William Kaufmann, 1982.

[Bailey90] R. Bailey, Functional Programming with Hope, Ellis Horwood, 1990.

[Bibel80] W. Bibel, "Syntax-directed, semantics-supported program synthesis", AI, 14:243-261, 1980.

[Biermann85] A.W. Biermann, "Automatic programming: A tutorial on formal methodologies", JSC, 1:119-142, 1985.

[BiGu83] A.W. Biermann y G. Guiho (dir.), Computer Program Synthesis Methodologies, D. Reidel Publishing Company, 1983.

[BiGuKo84] A.W. Biermann, G. Guiho e Y. Kodratoff (dir.), Automatic Program Construction Techniques, Macmillan Publishing Company, 1984.

[BiHö84] W. Bibel y K.M. Hörnig, "LOPS - A system based on a strategical approach to program synthesis", en [BiGuKo84], págs.69-89.

[Bird88] R. Bird, Introduction to Functional Programming, Prentice-Hall, 1988.

[BMPP89] F.L. Bauer, B. Möller, H. Partsch y P. Pepper, "Formal program construction by transformations - Computer-aided, Intuition-guided Programming", IEEE Trans. Soft. Eng., SE-15(2):165-180, febrero 89.

[BoMo75] R.S. Boyer y J.S. Moore, "Proving theorems about LISP functions", JACM, 21(1):129-144, enero 75.

[Brooks87] F.P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering", Computer, 20(4):10-19, abril 87.

[Broy83] M. Broy, "Program construction by transformations: A family tree of sorting programs", en [BiGu83], págs.1-49.

[BuDa77] R.M. Burstall y J. Darlington, "A transformation system for developing recursive programs", JACM, 24(1):44-67, enero 77.

[BuMaSa80] R.M. Burstall, D.B. MacQueen y D.T. Sannella, "HOPE: An experimental applicative language", Proc. 1980 Lisp Conf., Stanford, California, EE.UU., págs.136-143.

[Burstall69] R.M. Burstall, "Proving properties of programs by structural induction", CJ, 12:41-48, 1969.

[Burstall77] R.M. Burstall, "Design considerations for a functional programming language", Proc. Infotech State of the Art Conf., Copenhague, Dinamarca, págs.54-57, 1977.

[ClDa79] K.L. Clark y J. Darlington, "Algorithm classification through synthesis", CJ, 23(1):61-65, 1979.

[ClSi77] K.L. Clark y S. Sickel, "Predicate logic: A calculus for deriving programs", Proc. 5th IJCAI, Boston, Massachusetts, EE.UU., 1977, págs.419-420.

[CoFe82] P.R. Cohen y E.A. Feigenbaum (dir.), The Handbook of Artificial Intelligence, vol.III, Pitman Books, 1982.

[Darlington75] J. Darlington, "Aplications of program transformation to program synthesis", Proc. Int. Symp. on Proving and Improving Programs, Arc-Et-Sesans, Francia, 1-3 julio 75, págs.133-144.

[Darlington78] J. Darlington, "A synthesis of several sorting algorithms", Acta Informatica, 11:1-30, 1978.

[Darlington81a] J. Darlington, "An experimental program transformation and synthesis system", AI, 16:1-46, 1981. También en [RiWa86], págs.99-121.

[Darlington81b] J. Darlington, "The structured description of algorithm derivations", en Algorithmic languages, J.W. deBakker y H. van Vliet (dirs.), Elsevier North-Holland, 1981, págs.221-250.

[DCGMTTY88] P.J. Denning, D.E. Comer, D. Gries, M.C. Mulder, A. Tucker, A.J. Turner y P.R. Young, "Computing as a discipline", Final Report of the ACM Task Force on the Core of Computer Science, ACM Press, 1988. Versiones reducidas en CACM, 32(1):9-23, enero 89 y Computer, 22(2):63-70, febrero 89.

[DeLi86] D. DeGroot y G. Lindstrom, Logic Programming: Functions, Relations and Equations, Prentice-Hall, 1986.

[EiSa85] S. Eisenbach y C. Sadler, "Why functional programming", en Functional Programming: Languages, Tools and Arquitectures, Ellis Horwood, 1987, págs.9-17.

[Enderton72] H.B. Enderton, A Mathematical Introduction to Logic, Academic Press, 1972.

[Feather82] M.S. Feather, "A system for assisting program transformation", TOPLAS, 4(1):1-20, enero 82.

[Feather87] M.S. Feather, "A survey and classification of some program transformation approaches and techniques", en [Meertens87], págs.165-195.

[Fernández84] C. Fernández Chamizo, "Un sistema para la construcción automática de programas a partir de esquemas", tesis doctoral en ciencias físicas, Universidad Complutense de Madrid, 1984.

[FiHa88] A.J. Field y P.E. Harrison, Functional Programming, Addison-Wesley, 1988.

[GoMo87] J. Goguen y M. Moriconi, "Formalization in programming environments", Computer, 20(11): 55-64, noviembre 87.

[GrBa78] C. Green y D. Barstow, "On program synthesis knowledge", AI, 10:241-279, 1978. También en [RiWa86], págs.455-474.

[Green69] C. Green, "Application of theorem proving to problem solving", Proc. 1st IJCAI, Washington D.C., EE.UU., mayo 1969, págs.219-239.

[HaTä79] Å. Hansson y S-Å. Tärnlund, "A natural programming calculus", Proc. 6th IJCAI, Tokio, Japón, 1979.

[Henderson80] P. Henderson, Functional Programming: Application and Implementation, Prentice-Hall, 1980.

[Hogger81] C.J. Hogger, "Derivation of logic programs", JACM, 28(2):372-392, febrero 81.

[Knuth73] D. Knuth, The Art of Computer Programming, vol.3, Sorting and Searching, Addison-Wesley, 1973.

[Kott78] L. Kott, "About a transformational system: A theoretical study", Proc. 3rd Int. Symp. Prog., París, Francia, 1978, págs.232-247.

[Krieg84] B. Krieg-Brückner, "Language comparison by source-to-source translation", en [Pepper84], págs.299-304.

[MacLennan90] B.J. MacLennan, Functional Programming: Practice and Theory, Addison-Wesley, 1990.

[Mañas90] A.L. Mañas Molina, trabajo fin de carrera en realización, Facultad de Informática, Universidad Politécnica de Madrid.

[MaWa79] Z. Manna y R. Waldinger, "Synthesis: Dreams => Programs", IEEE Trans. Soft. Eng., SE-5(4):294-328, julio 79.

[MaWa80] Z. Manna y R. Waldinger, "A deductive approach to program synthesis", TOPLAS, 2(1):90-121, enero 80. También en [BiGuKo84], págs.33-68, y en [RiWa86], págs.3-34.

[MaWa81] Z. Manna y R. Waldinger, "Deductive synthesis of the unification algorithm", SCP, 1:5-48, 1981. También en [BiGu83], págs.251-307.

[MaWa85] Z. Manna y R. Waldinger, The Logical Basis for Computing Programming, vol.1, Addison-Wesley, 1985.

[MaWa86] Z. Manna y R. Waldinger, "Special relations in automated deduction", JACM, 33(1):1-59, enero 86.

[MaWa87] Z. Manna y R. Waldinger, "The origin of a binary search paradigm", SCP, 9:37-83, 1987.

[MaWa89] Z. Manna y R. Waldinger, "Program synthesis", material de trabajo de International Summer School on Logic, Algebra and Computation, Marktoberdorf, Alemania, 25 julio - 6 agosto 1989.

[MaWa90] Z. Manna y R. Waldinger, The Logical Basis for Computer Programming, vol.2, Addison-Wesley, 1990.

[Meertens87] L.G.L.T. Meertens (dir.), Program Specification and Transformation, North-Holland, 1987.

[Merrit85] S.M. Merrit, "An inverted taxonomy of sorting algorithms", CACM, 28(1):96-99, enero 85.

[Moreno89] J.J. Moreno Navarro, "Diseño, semántica e implementación de Babel: Un lenguaje que integra la programación funcional y lógica", tesis doctoral en informática, Universidad Politécnica de Madrid, junio 89.

[Murray82] N.V. Murray, "Completely non-clausal theorem proving", AI, 18:67-85, 1982.

[Nardi89] D. Nardi, "Formal synthesis of a unification algorithm by the deductive-tableau method", JLP, 7(1):1-43, julio 89.

[PaSt83] H. Partsch y R. Steinbrüggen, "Program transformation systems", Computing Surveys, 15(3):199-236, septiembre 83. También en [Agresti86], págs.189-226.

[Pepper84] P. Pepper (dir.), Program Transformation and Programming Environments, Springer-Verlag, 1984.

[Peyton87] S.L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall, 1987.

[Reade89] C. Reade, Elements of Functional Programming, Addison-Wesley, 1989.

[Reddy88] U.S. Reddy, "Transformational derivation of programs using the Focus system", Proc. ACM 3rd Symp. Soft. Development Environments, Boston, Massachusetts, EE.UU., 28-30 noviembre 88, págs.63-72.

[ReTe87] T. Reps y T. Teitelbaum, "Language processing in program editors", Computer, 20(11):29-40, noviembre 87.

[RiWa86] C. Rich y R.C. Waters (dir.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann Publishers Inc., 1986.

[RiWa88] C. Rich y R.C. Waters, "Automatic programming: Myths and prospects", Computer, 21(8):40-51, agosto 88.

[RiWa90] C. Rich y R.C. Waters, The Programmer's Apprentice, ACM Press, Addison-Wesley, 1990.

[Robinson65] J.A. Robinson, "A machine-oriented logic based on the resolution principle", JACM, 12(1):23-41, enero 65.

[Sannella88] D.T. Sannella, "A survey of formal software development methods", informe técnico ECS-LFCS-88-56, Department of Computer Science, University of Edinburgh, Gran Bretaña, julio 88.

[SaVe88] A. Sánchez Calle y J.A. Velázquez Iturbide, Apuntes de programación funcional, Departamento de Publicaciones, Facultad de Informática, Universidad Politécnica de Madrid, 1988.

[Scherlis81] W.L. Scherlis, "Program improvement by internal specialization", ACM Symp. Princ. Prog. Lang., 1981, págs.41-49.

[SlBr82] D. Sleeman y J. Brown (dir.), Intelligent Tutoring Systems, Academic Press, 1982.

[Smith82] D.E. Smith, "Derived preconditions and their use in program synthesis", Proc. 6th Conf. Automated Deduction, D.W. Loveland, (dir.), serie Lecture Notes on Computer Science 138, Springer-Verlag, págs.172-193.

[Smith83] D.E. Smith, "A problem reduction approach to program synthesis", Proc. 8th IJCAI, 1983, págs.32-36.

[Smith85a] D.E. Smith, "The design of divide and conquer algorithms", SCP, 5:37-58, 1985.

[Smith85b] D.E. Smith, "Top-down synthesis of divide-and-conquer algorithms", AI, 27:43-96, 1985. También en [RiWa86], págs.35-61.

[Smith85c] D.E. Smith, "Reasoning by cases and the formation of conditional programs", Proc. 9th IJCAI, Los Angeles, California, EE.UU., agosto 85, págs.215-218.

[Stickel81] M.E. Stickel, "A unification algorithm for associative-commutative functions", JACM, 28(3):423-434, julio 81.

[Stickel85] M.E. Stickel, "Automated deduction by theory resolution", JAR, 1:333-355, 1985.

[Sussman75] G.J. Sussman, A computer model of skill adquisition, American Elsevier, 1975.

[TaSa84] H. Tamaki y T. Sato, "Unfold/fold transformation of logic programs", Proc. 2nd Int. Logic Prog. Conf., Uppsala, 1984, págs.127-138.

[Traugott89] J. Traugott, "Deductive synthesis of sorting programs", JSC, 7:533-572, 1989.

[Velázquez90] J.A. Velázquez Iturbide, "Motivación de la programación transformacional", informe técnico FIM/58.1/LyS/90, Facultad de Informática, Universidad Politécnica de Madrid, mayo 90.

[Walther84] C. Walther, "Unification in many-sorted theories", Proc. 6th European Conference on Artificial Intelligence, Pisa, Italia, septiembre 84, págs.593-602.

[WiHo84] P.H. Winston y B.K.P. Horn, Lisp, Addison-Wesley, 2ª ed., 1984.

[Wikström87] Å. Wikström, Functional Programming Using Standard ML, Prentice-Hall, 1987.

[Wile83] D. Wile, "Program developments: Formal explanations of implementations", CACM, 26(11):902-911, noviembre 83. También en [RiWa86], págs.191-200, y en [Agresti86], págs.239-248.

[Wirth73] N. Wirth, "Program development by stepwise refinement", CACM, 14(4):221-227, abril 71.

[Wos85] L. Wos, "What is automated reasoning?", JAR, 1:6-9, 1987.

[Wos87] L. Wos, "Some obstacles to the automation of reasoning and the problem of redundant information", JAR, 3:81-90, 1987.

APENDICE A. TERMINOLOGIA Y NOTACION

Incluimos un breve recordatorio de los conceptos manejados. Se utiliza una lógica multigénero con géneros disjuntos; utilizamos la palabra tipo como sinónimo de género. Agrupamos los conceptos en varios apartados: sintaxis lógica, semántica lógica, simplificaciones y reescrituras, teorías lógicas, manejo de expresiones, relaciones binarias, sintaxis funcional. El lector puede encontrar exposiciones más detalladas en [Enderton72, MaWa85, MaWa86, MaWa90].

A.1. SINTAXIS DE LA LOGICA

Símbolo: Es un identificador de tipo, constante, función, predicado o conectiva. Un operador es un símbolo de función, predicado o conectiva.

Los símbolos de función y de predicado son unarios, es decir, tienen un solo parámetro. Cada símbolo de función tiene asociados dos tipos, uno del dominio y otro del rango. Cada símbolo de predicado tiene asociado un tipo de dominio.

Término: Es una expresión que denota un objeto. Un término se construye según las reglas:

- Una constante es un término con un tipo **T** asociado.
- Una variable es un término con un tipo **T** asociado.
- Si **f** es un símbolo de función de tipo $T_1 \rightarrow T_2$ y **t** es un término de tipo T_1, la aplicación
  f t
  es un término de tipo T_2.
- Si **F** es una sentencia y **s** y **t** son términos de tipo **T**, el condicional
  if F then s else t
  es un término de tipo **T**.

Proposición: Es una expresión que representa una relación entre objetos. Una proposición se construye según las reglas:

- Los símbolos de verdad
  true, false
  son proposiciones.
- Si **p** es un símbolo de predicado de tipo dominio **T**, y **t** es un término de tipo **T**,
  p t
  es una proposición.

Sentencia: Es una expresión que denota un valor de verdad. Una expresión se forma según las reglas:

- Una proposición es una sentencia.

- Si **F** es una sentencia, la negación
  (not F)
  es una sentencia.
- Si **F** y **G** son sentencias, la conjunción
  (F and G)
  es una sentencia. **F** y **G** se llaman los conjuntados de la conjunción.
- Si **F** y **G** son sentencias, la disjunción
  (F or G)
  es una sentencia. **F** y **G** se llaman los disjuntados de la disjunción.
- Si **F** y **G** son sentencias, la implicación
  (if F then G)
  es una sentencia. **F** se llama el antecedente y **G** se llama el consecuente de la implicación.
- Si **F** y **G** son sentencias, la equivalencia
  (F ≡ G)
  es una sentencia. **F** se llama el lado izquierdo y **G** se llama el lado derecho de la equivalencia.
- Si **F**, **G** y **H** son sentencias, la condicional
  (if F then G else H)
  es una sentencia. **F** se llama la cláusula if, **G** se llama la cláusula then y **H** se llama la cláusula else de la condicional.
- Si **x** es una variable de tipo **T** y **F** es una sentencia,
  ((∀ x:T) F), ((∃ x:T) F)
  son sentencias. Los prefijos **(∀ x:T)** y **(∃ x:T)** se llaman respectivamente cuantificador universal y cuantificador existencial. Representamos por **(... x:T)** un cuantificador, bien **(∀ x:T)** bien **(∃ x:T)**.

Nótese que algunas conectivas se representan mediante palabras clave en lugar de los símbolos lógicos usuales.

Notación informal: La siguiente notación informal no amplía el lenguaje de la lógica de predicados, sino que es un endulzamiento para conseguir mayor claridad.

Pueden suprimirse paréntesis en una sentencia (adoptando las prioridades usuales) y utilizar corchetes, las dos dimensiones y sangrado. También pueden utilizarse constantes, funciones y predicados especiales (en especial símbolos infijos), como **0**, **+** y **=**. Las funciones prefijas tienen mayor prioridad que las infijas. Además se utilizan los criterios usuales de prioridad entre funciones infijas. También puede expresarse una aplicación con el más usual formato con paréntesis **f(t)**. El término **f g t** representa la aplicación **f(g(t))**.

Notación informal de cuantificadores: Podemos agrupar informalmente varios cuantificadores en uno solo para reducir el tamaño de las sentencias de lógica de predicados de la manera siguiente:

- $(\forall\ x_1,\ldots,x_n{:}T)\ F$
  abrevia
  $(\forall\ x_1{:}T)\ \ldots\ (\forall\ x_n{:}T)\ F$

- $(\exists\ x_1,\ldots,x_n{:}T)\ F$
  abrevia
  $(\exists\ x_1{:}T)\ \ldots\ (\exists\ x_n{:}T)\ F$

- $(\forall\ x_1{:}T_1;\ldots;x_n{:}T_n)\ F$
  abrevia
  $(\forall\ x_1{:}T_1)\ \ldots\ (\forall\ x_n{:}T_n)\ F$

- $(\exists\ x_1{:}T_1;\ldots;x_n{:}T_n)\ F$
  abrevia
  $(\exists\ x_1{:}T_1)\ \ldots\ (\exists\ x_n{:}T_n)\ F$

En una lógica monogénero los cuantificadores relativizados son una notación informal con la misma sintaxis que los cuantificadores de una lógica multigénero. Los cuantificadores relativizados permiten que los cuantificadores afecten a subconjuntos limitados del dominio de interpretación. Se definen de la forma siguiente:

- $(\forall\ x{:}T)\ F$
  abrevia
  $(\forall\ x)\ \left[\begin{array}{l}\text{if } T(x)\\ \text{then } F\end{array}\right]$
- $(\exists\ x{:}T)\ F$
  abrevia
  $(\exists\ x)\ \left[\begin{array}{l}T(x)\\ \text{and}\\ F\end{array}\right]$

donde **T** es un símbolo de predicado de género y **F** es una sentencia.

Expresión: Una expresión es un término o una sentencia.

Variables libres y ligadas: Sea una variable **x** de tipo **T** y una expresión **I**. Una aparición de **x** está ligada en **I** si está dentro del ámbito de un cuantificador **(... x:T)** en **I**. Una aparición de **x** está libre en **I** si no está dentro del ámbito de ningún cuantificador **(... x:T)** en **I**. Una variable está ligada en una expresión **I** si existe al menos una aparición ligada de **x** en **I**, y libre si existe al menos una aparición libre de **x** en **I**.

Clases de expresiones: Una expresión es básica si no contiene variables. La expresiones que intervienen en la formación de una expresión son sus subexpresiones. Una subexpresión propia es distinta de la expresión completa. De manera similar se definen los subtérminos y las subsentencias de una expresión.

Una aparición de una subexpresión **I** está ligada en una expresión **J** si existe alguna aparición de alguna variable **x** libre en **I** tal que dicha aparición está ligada en **J**. Una aparición de una subexpresión **I** está libre en una expresión **J** si cada aparición de una variable libre en **I** también es libre en **J**.

Clases de sentencias: Una sentencia es cerrada si no tiene ninguna aparición libre de ninguna variable. Dos sentencias están mutuamente estandarizadas o estandarizadas entre sí si las sentencias no contienen ninguna variable común.

Cierres universal y existencial: Supóngase que la lista completa de variables libres de una sentencia **F** es $x_1,\ldots,x_n$ de tipos respectivos $T_1,\ldots,T_n$. Entonces el cierre universal de **F**, representado **(∀*) F**, es la sentencia cerrada

$$(\forall\ x_1{:}T_1;\ldots;x_n{:}T_n)\ F$$

y el cierre existencial de **F**, representado **(∃*) F**, es la sentencia cerrada

$$(\exists\ x_1{:}T_1;\ldots;x_n{:}T_n)\ F$$

Polaridad de una sentencia: La polaridad de una subsentencia **E** en una sentencia **S** es un mecanismo sintáctico que indica cómo contribuye la verdad de **E** a la verdad de **S**. Una subsentencia puede tener polaridad positiva (representado E^+), negativa (E^-) o ambas ($E^\pm$) en una sentencia. Dos subsentencias de una sentencia tienen polaridades opuestas si una tiene polaridad positiva y la otra negativa; la opuesta de una polaridad π se representa $-\pi$.

Una aparición de una subsentencia **E** de una sentencia **S** tiene polaridad estrictamente positiva en **S** si **E** tiene polaridad positiva, pero no negativa. Análogamente se define la polaridad estrictamente negativa.

Reglas de asignación de polaridad: Básicamente, una aparición de una subsentencia en una sentencia es positiva (o negativa) si está en el ámbito de un número par (o impar) de negaciones, tanto explícitas como implícitas.

Dada una sentencia **S**, asignamos una polaridad a cada subsentencia de **S** de acuerdo con las siguientes reglas de asignación de polaridad:

- S^+
- $[\text{not } F]^\pi \equiv> \text{not } F^{-\pi}$
- $[F \text{ and } G]^\pi \equiv> F^\pi \text{ and } G^\pi$
- $[F \text{ or } G]^\pi \equiv> F^\pi \text{ or } G^\pi$
- $[\text{if } F \text{ then } G]^\pi \equiv> \text{if } F^{-\pi} \text{ then } G^\pi$
- $[F \equiv G] \equiv> F^\pm \equiv G^\pm$
- $[\text{if } F \text{ then } G \text{ else } H]^\pi \equiv> \text{if } F^\pm \text{ then } G^\pi \text{ else } G^\pi$
- $[(\forall\ x{:}T)\ F]^\pi \equiv> (\forall\ x)\ F^\pi$
- $[(\exists\ x{:}T)\ F]^\pi \equiv> (\exists\ x)\ F^\pi$
- $[\text{if } F \text{ then } s \text{ else } t]^\pi \equiv> \text{if } F^\pm \text{ then } s \text{ else } t.$

Para el uso de estas reglas, se comienza con la sentencia **S** y sucesivamente asignamos una polaridad a cada uno de sus componentes sintácticos, hasta que todas las subsentencias tienen asignada una polaridad.

Fuerza de un cuantificador: La fuerza de un cuantificador en una sentencia es un mecanismo sintáctico que indica el

papel que el cuantificador juega en la sentencia. Un cuantificador **(... x:T)** de una subsentencia **(... x:T) F** de una sentencia **S** puede tener fuerza universal (representado **(... x:T)$^{\forall}$ F**), existencial (**(... x:T)$^{\exists}$ F**) o ambas (**(... x:T)$^{\forall\exists}$ F**) en **S**.

Decimos que una aparición de un cuantificador tiene fuerza universal estricta si tiene fuerza universal estricta pero no existencial. Análogamente se define la fuerza existencial estricta.

Reglas de asignación de fuerza: Básicamente, una aparición de un cuantificador tiene fuerza universal (o existencial) en una sentencia si es un cuantificador universal en el ámbito de un número par (o impar) de negaciones.

La fuerza de una aparición de una subsentencia **(... x:T) F** en una sentencia **S** se determina mediante las reglas:

- La aparición del cuantificador tiene fuerza universal en **S** si:
  es un cuantificador universal y **(... x:T) F** tiene polaridad positiva en **S**, es decir
  $[(\forall\ x{:}T)\ F]^{+} \equiv> (\forall\ x{:}T)^{\forall}\ F$
  o
  es un cuantificador existencial y **(... x:T) F** tiene polaridad negativa en **S**, es decir
  $[(\exists\ x{:}T)\ F]^{-} \equiv> (\exists\ x{:}T)^{\forall}\ F.$

- La aparición del cuantificador tiene fuerza existencial en **S** si:
  es un cuantificador existencial y **(... x:T) F** tiene polaridad positiva en **S**, es decir
  $[(\exists\ x{:}T)\ F]^{+} \equiv> (\exists\ x{:}T)^{\exists}\ F$
  o
  es un cuantificador universal y **(... x:T) F** tiene polaridad negativa en **S**, es decir
  $[(\forall\ x{:}T)\ F]^{-} \equiv> (\forall\ x{:}T)^{\exists}\ F.$
- La aparición del cuantificador tiene ambas fuerzas si **(... x:T) F** tiene ambas polaridades, es decir
  $[(\ldots\ x{:}T)\ F]^{\pm} \equiv> (\ldots\ x{:}T)^{\forall\exists}\ F.$

Proposición de reemplazamiento universal y existencial: Sean una variable **x** y un término **t**, ambos de tipo **T** y tales que **x** no aparece libre en **t**, y una sentencia **F[x]**. Entonces,
(∀ x:T) if x=t then F[x] es equivalente a **F[t]**
y
(∃ x:T) x=t and F[x] es equivalente a **F[t]**.

Proposición de sustitutividad por equivalencia: Dadas tres sentencias cerradas **G**, **H** y **F<G>**, si **G** es equivalente a **H**, **F<G>** es equivalente a **F<H>**.

A.2. SIMPLIFICACIONES Y REESCRITURAS

A continuación incluimos un catálogo de simplificaciones y

reescrituras del sistema lógico [MaWa90]. Algunas de las reescrituras son innecesarias si incluimos un algoritmo de unificación conmutativo-asociativo [Stickel81] que tome en consideración estas propiedades en las conectivas.

Simplificaciones true-false:

Negación:

not true	=>	false	(not-true)
not false	=>	true	(not-false)

Conjunción:

F and true	=>	F	(and-true)
true and F	=>	F	(true-and)
F and false	=>	false	(and-false)
false and F	=>	false	(false-and)

Disjunción:

F or true	=>	true	(or-true)
true or F	=>	true	(true-or)
F or false	=>	F	(or-false)
false or F	=>	F	(false-or)

Implicación:

if true then G	=>	G	(if-true)
if false then G	=>	true	(if-false)
if F then true	=>	true	(then-true)
if F then false	=>	not F	(then-false)

Equivalencia:

F ≡ true	=>	F	(sii-true)
true ≡ F	=>	F	(true-sii)
F ≡ false	=>	not F	(sii-false)
false ≡ F	=>	not F	(false-sii)

Condicional:

if true then G else H	=>	G	(cond-if-true)
if false then G else H	=>	H	(cond-if-false)
if F then true else H	=>	F or H	(cond-then-true)
if F then false else H	=>	(not F) and H	(cond-then-false)
if F then G else true	=>	if F then G	(cond-else-true)
if F then G else false	=>	F and G	(cond-else-false)

Término condicional:

if true then s else t	=>	s	(term-cond-true)
if false then s else t	=>	t	(term-cond-false)

Otras simplificaciones:

Negación:

not (not F)	=>	F	(not-not)
not ((not F) and (not G))	=>	F or G	(not-and)
not ((not F) or (not G))	=>	F and G	(not-or)
not (if F then (not G))	=>	F and G	(not-if)

```
        not (F ≡ (not G))  =>  F ≡ G                    (not-sii)
        not (if F then (not G) else (not H))
             => if F then G else F                      (not-cond)
```

Conjunción:

```
              F and F  =>  F                             (and-dos)
        F and (not F)  =>  false                         (and-not)
        (not F) and F  =>  false                         (not-and)
```

Disjunción:

```
              F or F  =>  F                              (or-dos)
        F or (not F)  =>  true                           (or-not)
        (not F) or F  =>  true                           (not-or)
```

Implicación:

```
                    if F then F  =>  true                (if-dos)
              if (not F) then F  =>  F                   (if-not)
              if F then (not F)  =>  not F             (then-not)
        of (not G) then (not F)
             =>  if F then G                      (contrapositivo)
```

Condicional:

```
        if F then G else G  =>  G                        (cond-dos)
        if (not F) then G else H
             =>  if F then H else G                      (cond-not)
```

Cuantificador redundante:

```
        (∀ x:T) F  =>  F
             si x no está libre en F                       (todo)
        (∃ x:T) F  =>  F
             si x no está libre en F                     (existe)
```

Término condicional:

```
        if F then t else t  =>  t                   (term-cond-dos)
```

Igualdad:

```
        t = t  =>  true                               (term-igual)
```

Reescrituras:

Negación:

```
            not (F and G)  <=>  (not F) or (not G)    (not-and)
             not (F or G)  <=>  (not F) and (not G)    (not-or)
        not (if F then G)  <=>  F and (not G)          (not-if)
              not (F ≡ G)  <=>  F ≡ (not G)           (not-sii)
    not (if F then G else H)
         <=>  if F then (not G) else (not H)         (not-cond)
```

Eliminación:

```
    if F then G <=>  (not F) or G                        (if-or)
       F ≡ G  <=> (if F then G) and (if G then F)      (sii-and)
       F ≡ G  <=> (F and G) or ((not F) and (not G))   (sii-or)
    if F then G else H
         <=> (if F then G) and (if (not F) then H)   (cond-and)
    if F then G else H
         <=>  (F and G) or ((not F) and H)           (cond-or)
```

Conmutatividad:

```
          F and G  <=>  G and F                    (and)
           F or G  <=>  G or F                      (or)
            F ≡ G  <=>  G ≡ F                      (iff)
```

Asociatividad:

```
        (F and G) and H  <=>  F and (G and H)       (and)
          (F or G) or H  <=>  F or (G or H)          (or)
            (F ≡ G) ≡ H  <=>  F ≡ (G ≡ H)           (sii)
```

Distributividad:

```
        F and (G or H)  <=>  (F and G) or (F and H)  (and-or)
        F or (G and H)  <=>  (F or G) and (F or H)   (or-and)
```

Reordenamiento de cuantificadores:

```
        (∀ x:T) (∀ y:T') F  <=>  (∀ y:T') (∀ x:T) F    (todo)
        (∃ x:T) (∃ y:T') F  <=>  (∃ y:T') (∃ x:T) F   (algún)
```

Dualidad de cuantificadores:

```
        not (∀ x:T) F  <=>  (∃ x:T) (not F)        (not-todo)
        not (∃ x:T) F  <=>  (∀ x:T) (not F)       (not-algún)
```

Manipulación de cuantificadores:

```
    (∀ x:T) [F and G]  <=> (∀ x:T) F and (∀ x:T) G  (todo-and)
    (∃ x:T) [F or G]   <=>  (∃ x:T) F or (∃ x:T) G   (algún-or)
    (∃ x:T) [if F then G]
              <=> if (∀ x:T) F then (∃ x:T) G       (algún-if)
```

Manipulación de condicional:

```
    p(x,if F then s else t,y)
         <=>  if F then p(x,s,y) else p(x,t,y)     (predicado)
    f(x,if F then s else t,y)
         <=>  if F then f(x,s,y) else f(x,t,y)       (función)
```

A.3. SEMANTICA DE LA LOGICA

Interpretación: Sean **n** conjuntos D_i no vacíos cualesquiera. Una interpretación **I** sobre los dominios D_i asigna valores a un conjunto de símbolos como sigue:

- A cada tipo T_i ($1 \le i \le n$) del conjunto de símbolos le asigna un dominio D_i.
- A cada constante **a** de tipo T_i le asigna un elemento a_I de D_i.
- A cada variable **x** de tipo T_i, un elemento x_I de D_i.
- A cada símbolo de función **f** de tipo $D_i \to D_j$ ($1 \le i,j \le n$), una función f_I; la función f_I está definida sobre un argumento **d** de D_i, y su valor f_I **d** pertenece a D_j.
- A cada símbolo de predicado **p** de dominio D_i, una relación p_I; la relación f_I está definida sobre un argumento **d** de D, y su valor p_I **d** es cierto o falso.

Se dice que una interpretación **I** es una interpretación de una expresión **E** si **I** asigna un valor a cada símbolo libre de **E**.

Valor bajo una interpretación: Dada una interpretación **I** de una expresión **E**, podemos determinar su valor. Para una sentencia, este valor es un valor de verdad (cierto o falso); para un término de término **T**, este valor es un objeto del dominio **D** asociado a **T** por la interpretación. Dada una expresión **E** y una interpretación **I**, el valor de **E** bajo **I** se calcula aplicando repetidamente las siguientes reglas semánticas:

- El valor de una constante
  a
  de tipo T_i es el elemento a_I del dominio D_i.
- El valor de una variable
  x
  de tipo T_i es el elemento x_I del dominio D_i.
- El valor de una aplicación
  f t
  es el elemento f_I **d** del dominio D_j, donde f_I es la función asignada a **f** y **d** es el valor del dominio D_i asignado al término **t** bajo **I**.
- El valor del término condicional
  if F then s else t
  es el valor del término **s** si la sentencia **F** es cierta y el valor del término **t** si **F** es falsa, bajo **I**.
- El valor de los símbolos de verdad
  true, false
  son los valores de verdad cierto y falso respectivamente.
- El valor de una proposición
  p t
  es el valor de verdad p_I **d** (bien cierto bien falso) donde p_I es la relación asignada a **p** y **d** es el valor del término **t** bajo **I**.
- El valor de la negación
  not F
  es cierto si la sentencia **F** es falsa y falso si **F** es cierta.
- El valor de la conjunción
  F and G
  es cierto si **F** y **G** son ambas ciertas y falso en caso contrario.
- El valor de la disjunción
  F or G
  es cierto si ora **F** ora **G** es cierto y falso en caso contrario.
- El valor de la implicación
  if F then G
  es cierto si **F** es cierto o **G** es falso y falso en caso contrario.
- El valor de la equivalencia
  F ≡ G
  es cierto si los valores de verdad de **f** y **G** son iguales y falso en caso contrario.

- El valor de la condicional
  if F then G else H
  es el valor de **G** si **F** es cierta y es el valor de **H** si **F** es falsa.
- El valor de la cuantificación universal
  (V x:T_i) F[x]
  es cierto bajo una interpretación **I** si la ampliación de la interpretación **I** con la asignación de un elemento **d** cualquiera del dominio D_i a la variable **x** hace que el valor de **F[x]** sea cierto, y falso en caso contrario.
- El valor de la cuantificación existencial
  (∃ x:T_i) F[x]
  es cierto bajo una interpretación **I** si existe algún elemento **d** del dominio D_i tal que la ampliación de **I** con la asignación de **d** a la variable **x** hace que el valor de **F[x]** sea cierto, y falso en caso contrario.

Validez: Una sentencia cerrada **F** es válida si es cierta bajo cada interpretación de **F**. Una sentencia válida se llama un teorema. Una sentencia cerrada **F** es satisfactible si es cierta bajo alguna interpretación de **F**. Una sentencia **F** es contradictoria (o insatisfactible) si es falsa bajo todas las interpretaciones de **F**. Un conjunto de sentencias cerradas $F_1,F_2,...$ es consistente si existe alguna interpretación de $F_1,F_2,...$ bajo la que todas las F_i son ciertas.

Implicación: Las sentencias $F_1,...,F_n$ implican una sentencia **G** si, bajo cada interpretación **I** de $F_1,...,F_n$ y **G**, si $F_1,...,F_n$ son ciertas bajo **I**, entonces **G** es cierta bajo **I**. Dos sentencias **F** y **G** son equivalentes si, bajo cada interpretación de **F** y **G**, **F** tiene el mismo valor de verdad que **G**.

A.4. TEORIAS LOGICAS

Teoría: Una teoría es un conjunto **T** de sentencias que está cerrado bajo la implicación lógica, es decir si **T** implica una sentencia **F** entonces **F** pertenece a **T**.

Una teoría consta de un lenguaje y un conjunto de sentencias (llamadas axiomas). El lenguaje de una teoría es el lenguaje de lógica de predicados restringido a cierto vocabulario o conjunto de símbolos de constante, función y predicado. Los términos, sentencias y expresiones de una teoría son los términos, sentencias y expresiones de lógica de predicados cuyos símbolos de constante, función y predicado pertenecen al vocabulario de la teoría.

Los axiomas de una teoría son un conjunto **A** de sentencias cerradas $A_1,A_2,...$ de la teoría tales que **T** es precisamente el conjunto de sentencias implicado por **A**. También se dice que una teoría se define por sus axiomas.

Interpretación y validez en una teoría: Una interpretación **I** es un modelo de una teoría si todo axioma A_i de la teoría es cierto bajo **I**. Una teoría es consistente si existe al menos un

modelo de la teoría.

Una sentencia cerrada **F** de una teoría es válida en la teoría si **F** es cierta bajo cada modelo de la teoría. Una sentencia válida en una teoría se llama un teorema de la teoría. Una sentencia **F** implica una sentencia **G** de la teoría si, siempre que **F** es cierta bajo un modelo de la teoría, **G** es también cierta bajo el modelo. Dos sentencias **F** y **G** son equivalentes en la teoría si **F** y **G** tienen el mismo valor de verdad bajo cualquier modelo de la teoría.

A.5. MANEJO DE EXPRESIONES

Reemplazamiento: Sean I, J y K expresiones, siendo I y J simultáneamente sentencias o términos del mismo tipo. Si escribimos K como K[I], entonces K[J] denota la expresión obtenida tras reemplazar en K[I] cada aparición de I por K. Esta operación se llama reemplazamiento total.

Si escribimos K como K<I>, entonces K<J> denota la expresión obtenida al reemplazar en K<I> ciertas apariciones (cero, una o más) de I por J.

Obsérvese que el reemplazamiento total produce un resultado único, mientras que el reemplazamiento parcial puede producir varias expresiones. En ninguno de los dos reemplazamientos se exige que I aparezca en K.

Reemplazamiento múltiple: Puede ampliarse la definición de reemplazamiento para permitir reemplazar varias subexpresiones simultáneamente.

Sean $I_1,\ldots,I_n,J_1,\ldots,J_n$ y K expresiones, donde las I_i son distintas y, para cada i, I_i y J_i son simultáneamente sentencias o términos del mismo tipo. Si escribimos K como $K[I_1,\ldots,I_n]$, entonces $K[J_1,\ldots,J_n]$ denota la expresión obtenida al reemplazar simultáneamente en K cada aparición de cada expresión I_i por la expresión correspondiente J_i. Esta operación se llama reemplazamiento total múltiple.

Puede haber un conflicto si una expresión I_i es una subexpresión de otra expresión I_j. En este caso, se sustituye la subexpresión más externa, es decir I_j, por su expresión correspondiente, es decir J_j.

Si escribimos K como $K<I_1,\ldots,I_n>$, entonces $K<J_1,\ldots,J_n>$ denota la expresión obtenida al reemplazar simultáneamente en K ciertas apariciones (cero, una o más) de algunas expresiones I_i por la expresión correspondiente J_i.

Reemplazamiento parcial con polaridad: Sean las expresiones I, J y $K<I^+>$. Entonces $K<J^+>$ denota el resultado de reemplazar en $K<I^+>$ ciertas apariciones (cero, una o más) estrictamente positivas de I por J.

Análogamente, sean las expresiones I, J y K<I$^-$>. Entonces K<J$^-$> denota el resultado de reemplazar en K<I$^-$> ciertas apariciones (cero, una o más) estrictamente negativas de I por J.

De manera análoga a la generalización realizada anteriormente también definimos el reemplazamiento múltiple con polaridad.

Sustitución: Se denomina sustitución a un caso particular de reemplazamiento, el de variables por términos. Dadas las variables $x_1,\ldots,x_n$, diferentes entre sí, y los términos $t_1,\ldots,t_n$, una sustitución

$$\theta: \{ x_1<-t_1, \ldots , x_n<-t_n \}$$

es un conjunto de pares de reemplazamiento $x_i<-t_i$ donde x_i y t_i tienen el mismo tipo. La sustitución vacía {} es el conjunto que no contiene ningún par de reemplazamiento.

Dada una sustitución θ y una expresión e, denotamos por $e\theta$ la expresión obtenida al aplicar θ a e, es decir, al sustituir simultáneamente, para cada par de reemplazamiento $x_i<-t_i$ de θ, cada aparición en e de la variable x_i por la expresión t_i. También se dice que $e\theta$ es una instancia de e.

Sustitución más general: La composición $\theta\mu$ de dos sustituciones θ y μ es una sustitución que cumple la propiedad de que su aplicación a una expresión e produce el mismo resultado que aplicar primero la sustitución θ y después μ, es decir

$$e(\theta\mu) = (e\theta)\mu$$

Una sustitución θ es más general que una sustitución θ' si existe una sustitución μ tal que $\theta\mu=\theta'$.

Unificador: Se dice que una sustitución es un unificador de dos expresiones e y f si $e\theta=f\theta$.

Se dice que una sustitución θ es un unificador más general de dos expresiones e y f si θ es un unificador de e y f y si θ es más general que ningún otro unificador de e y f. Existe un algoritmo de unificación [Walther84] que, dado un par de expresiones, determina si es unificable y en caso positivo, produce un unificador más general. El algoritmo puede ampliarse para que considere la conmutatividad y la asociatividad de las funciones [Stickel81].

Podemos extender la noción de unificador a una lista e de expresiones $e_1,\ldots,e_n$. Se dice que una sustitución θ es un unificador simultáneo de la lista e si $e_1\theta=\ldots=e_n\theta$.

Sustitución y reemplazamiento: A veces es conveniente usar conjuntamente las nociones de reemplazamiento y sustitución. Sean las expresiones I, J y K, donde I y J son simultáneamente sentencias o términos, y sea la sustitución θ. Si escribimos K como K[I], entonces Kθ[J] denota la expresión resultante de reemplazar en Kθ cada aparición de Iθ por J. Si escribimos K como K<I>, entonces Kθ<J> denota la expresión resultante de

reemplazar en **K⊖** ciertas apariciones (cero, una o más) de **I⊖** por **J**.

Réplicas: Dos expresiones **I** y **J** son réplicas si ambas expresiones son instancias de una sentencia **K** más general y ninguna (ni **I** ni **J**) es una instancia de la otra.

A.6. RELACIONES BINARIAS

Relación: Un símbolo de relación es un símbolo de predicado o de conectiva. (Las conectivas lógicas se consideran relaciones sobre el conjunto de valores booleanos {true, false}.)

Sean **p** y **q** dos relaciones sobre n-tuplas. Se dice que **p** implica **q** si, para expresiones $I_1,\ldots,I_n$ cualesquiera,

if $p(I_1,\ldots,I_n)$ then $q(I_1,\ldots,I_n)$

es válida.

Relación binaria: Es una relación << definida sobre pares.

La inversa >> de << se define, para expresiones **I**, **J** cualesquiera, como

$$I>>J \equiv J<<I$$

Dadas dos relaciones $<_1$ y $<_2$ definidas sobre pares de elementos de tipo **T**, $<_1$ es una subrelación de $<_2$ si se cumple:

$$(\forall\ x,y:T)\ [\ \text{if } x<_1y\ \text{ then } x<_2y\ ]$$

Relación bien fundada: Es una relación binaria definida sobre pares de elementos de un tipo **T** cualquiera, tal que no existen secuencias decrecientes infinitas, es decir, no existen secuencias de valores $x_1,x_2,\ldots$ de tipo **T** tal que cumplan $x_1>>x_2>>\ldots$.

Proposición de subrelación bien fundada: Dadas dos relaciones $<_1$ y $<_2$ tales que $<_1$ es una subrelación de $<_2$,

si $<_2$ es una relación bien fundada,

entonces $<_1$ también es una relación bien fundada.

Monotonicidad: Una subexpresión es monotónicamente creciente (o simplemente creciente) en una expresión si al reemplazar la subexpresión por otra mayor, se obtiene una expresión mayor. La subexpresión es monotónicamente decreciente si el mismo reemplazamiento produce una expresión menor. Las palabras "mayor" y "menor" siempre son relativas a dos relaciones $<_1$ y $<_2$.

Polaridad: Es un mecanismo sintáctico que identifica la monotonicidad de una expresión respecto a dos relaciones $<_1$ y $<_2$. Si una subexpresión **I** es monotónicamente creciente, decreciente, ambas propiedades o ninguna dentro de un expresión **J** respecto a dos relaciones $<_1$ y $<_2$, se dice que tiene una polaridad positiva (representada $J[I^{+<1<2}]$), negativa

($J[I^{-<1<2}]$), ambas polaridades ($J[I^{\pm<1<2}]$) o ninguna respecto a $<_1$ y $<_2$ en J.

Si la relación $<_2$ es la conectiva **if-then**, simplemente hablamos de polaridad de una subexpresión I en una expresión J respecto a $<_1$ (representado $J[I^{\pi<1}]$, donde π es la polaridad respecto a $<_1$). Si además la relación $<_1$ es la conectiva **if-then**, solamente hablamos de la polaridad de la subsentencia F en la sentencia G (representado $G[F^{\pi}]$, donde π es la polaridad), como se expone en el apartado A.1.

Polaridad de un operador: Sea O un operador con un dominiode n-tuplas y $<_1$, $<_2$ dos relaciones binarias. Entonces,

- O es positivo sobre su i-ésimo argumento respecto a $<_1$ y $<_2$ si, para expresiones $I_1,\ldots,I_n,J$ cualesquiera, la sentencia
  if $I_1<_1J$
  then $O(I_1,\ldots,I_i,\ldots,I_n) <_2 O(I_1,\ldots,J,\ldots,I_n)$
  es válida.
- O es negativo sobre su i-ésimo argumento respecto a $<_1$ y $<_2$ si, para expresiones $I_1,\ldots,I_n,J$ cualesquiera, la sentencia
  if $I_1<_1J$
  then $O(I_1,\ldots,J,\ldots,I_n) <_2 O(I_1,\ldots,I_i,\ldots,I_n)$
  es válida.

Puede comprobarse que cualquier relación es tanto positiva como negativa sobre cada uno de sus argumentos respecto a la relación de igualdad = (según la propiedad de sustitutividad en predicados de la igualdad). Además, cada función es tanto positiva como negativa sobre cada uno de sus argumentos respecto a = y = (según la propiedad de sustitutividad funcional de la igualdad).

Polaridad de una subexpresión:
Sean $<_1$ y $<_2$ dos relaciones binarias. Entonces:

- Una expresión I es positiva en I misma respecto a $<_1$ y $<_2$ si $<_1$ implica $<_2$.
- Una expresión I es negativa en I misma respecto a $<_1$ y $<_2$ si $<_1$ implica $>_2$.

Sea O un operador definido sobre n-tuplas y sean $I_1,\ldots,I_n$ expresiones cualesquiera. Considérese una aparición de J en una expresión I_i. Entonces:

- La aparición de J es positiva en $O(I_1,\ldots,I_n)$ respecto a $<_1$ y $<_2$ si existe una relación binaria << tal que:
  la polaridad de la aparición de J en I_i respecto a $<_1$ y <<
  es la misma que
  la polaridad de O sobre su i-ésimo argumento respecto a << y $<_2$.
- La aparición de J es negativa en $O(I_1,\ldots,I_n)$ respecto a $<_1$ y $<_2$ si existe una relación binaria << tal que:
  la polaridad de la aparición de J en I_i respecto a $<_1$ y <<
  es la contraria que

la polaridad de O sobre su i-ésimo argumento respecto a << y $<_2$.

Además, si O no tiene polaridad sobre su i-ésimo argumento o si J no tiene polaridad en I_i, entonces J no tiene polaridad en $O(I_1,...,I_n)$. Por otro lado, si J tiene ambas polaridades en I_i y O tiene alguna polaridad sobre su i-ésimo argumento, o si O tiene ambas polaridades sobre su i-ésimo argumento y J tiene alguna polaridad en I_i, entonces J tiene ambas polaridades en $O(I_1,...,I_n)$.

Proposición de reemplazamiento de polaridad:
Para cualquier relación binaria << y sentencia $F<I^{+<<},J^{-<<}>$, la sentencia

if I<<J
then if $F<I^{+<<},J^{-<<}>$
then $F<J^{+<<},I^{-<<}>$

es válida. Aquí $F<J^{+<<},I^{-<<}>$ es el resultado de reemplazar en $F<I^{+<<},J^{-<<}>$ ciertas apariciones positivas de I por J y ciertas apariciones negativas de J por I.

A.7. SINTAXIS FUNCIONAL

Función total y parcial: Una función se llama parcial si existe algún elemento de su tipo dominio que no tiene asociado ningún elemento del tipo rango.

Programa funcional: Es un texto formado por un conjunto de declaraciones de tipo y declaraciones de función.

Declaración de tipo: Es una construcción que declara el modo de formar sintácticamente los valores de un tipo. Una declaración de tipo tiene el formato:

data T == C_1 T_1 ++ ... ++ C_m T_m ;

donde T es un identificador del nuevo tipo,
T_i ($1 \le i \le m$), opcional, es un tipo de n-tuplas ($n \ge 1$) formadas por elementos de tipos ya definidos o de tipo T,
C_i ($1 \le i \le m$) es bien un símbolo de constante de tipo T bien un símbolo de función de tipo dominio T_i y tipo rango T.

Cada símbolo C_i se llama un constructor. Una función constructora no se define con una regla de cálculo.

Se dice que un tipo T es recursivo si alguna función constructora C_i de su definición tiene como tipo dominio T_i una tupla donde algún elemento es de tipo T.

Término funcional: Es una expresión utilizable en un programa funcional. Un término funcional se forma según las reglas:

- Un término de la lógica de tipo **T** es un término funcional de tipo **T**.

- Si t_1 es un término funcional de tipo T_1 formado exclusivamente por variables y constructores, y t_2 y t_3 son dos términos funcionales cualesquiera de tipos respectivos T_1 y T_2, la definición local:

  let t_1 == t_2
  in t_3

  o

  t_3
  where t_1 == t_2

  es un término funcional de tipo T_2.

  t_1 se llama el término cualificado, t_2 se llama el término cualificador y t_3 se llama el término resultado.

Los conceptos de término básico y subtérmino se definen igual que en la lógica de predicados.

Valor de una expresión básica: Una expresión básica, bien un término funcional bien una sentencia, puede reescribirse es una expresión igual o equivalente, respectivamente, en cualquier modelo de la teoría. La expresión final se llama el valor de la expresión inicial.

Valores de un tipo de datos: El conjunto de valores de un tipo **T** está formado por los términos básicos construibles a partir de sus constructores constantes más el conjunto de aplicaciones de cada función constructora C_i a todos los valores de su tipo dominio T_i. Este uso de los constructores como base sintáctica para representar los valores de los tipos de datos hace que los tipos de datos de datos se llmen álgebras de palabras.

Existe un método de cálculo del conjunto de valores de un tipo (recursivo) **T**. En primer lugar se forma el conjunto de constantes constructoras del tipo. Después se halla el conjunto de aplicaciones de sus funciones constructoras a tuplas de valores ya hallados. Si aparece involucrado algún tipo distinto de **T**, se toman todos sus valores posibles. Se repite sucesivas veces este proceso de forma tal que en cada iteración se aplican las funciones constructoras a tuplas de valores obtenidos previamente y se eliminan los valores ya existentes o repetidos. El conjunto de valores del tipo es la unión de todos estos conjuntos.

Símbolo predefinido: Es un símbolo de operador que, dados unos operandos básicos, forma una expresión básica con un valor asociado de manera inmediata. Todos los símbolos de predicado y conectiva se consideran predefinidos.

Valor de una sentencia básica: Una sentencia básica tiene como valor los símbolos de verdad **true** o **false**.

Valor de un término funcional básico: Un término funcional

básico tiene como valor otro término funcional básico formado exclusivamente por constantes y funciones constructoras. El valor de un término funcional básico se calcula según las reglas:

- El valor de una constante
  c
  es la misma constante **c**.
- El valor de una aplicación de función predefinida
  f t
  es un valor **v** dependiente de la función **f** particular.
- El valor de una aplicación de función constructora
  C t
  es el término **C t'**, donde **t'** es el valor de **t**.
- El valor de una aplicación de función no predefinida y no constructora
  f t
  donde **t** tiene un valor **t'** y **t'[ē]** es unificable con el término **t'[x̄]** del lado izquierdo de una ecuación
  --- f t'[x̄] <= s[x̄] ;
  es el valor del término **s[ē]**.
- El valor de un término condicional
  if F then s else t
  es el valor de **s** si la sentencia básica **F** tiene un valor **true** y es el valor de **t** si tiene un valor **false**.
- El valor de una definición local
  let t_1 == t_2
  in t_3
  donde t_2 tiene un valor $t_1[\bar{e}]$ unificable con el término cualificado $t_1[\bar{x}]$ y t_3 tiene la forma $t_3[\bar{x}]$, es el valor del término $t_3[\bar{e}]$.

Patrón: Es un término funcional formado exclusivamente por variables y constructores y utilizado en algunas construcciones sintácticas para descomponer un valor en sus componentes.

Declaración de función: Es una construcción sintáctica que define una función nueva. Deben declararse todas las funciones excepto las funciones constructoras y las funciones predefinidas. Una declaración de función consta de una declaración de tipo y un conjunto de ecuaciones.

Declaración de tipo de función: Es una construcción sintáctica que declara el símbolo **f** de una función y su tipo, consistente en un tipo dominio **D** y un tipo rango **R**. El formato de una declaración de tipo de función es:

```
dec f : D -> R ;
```

Ecuación: *Es una regla de cálculo que expresa el valor de la función cuando el valor de su parámetro se ha construído de cierta forma, es decir, coincide con cierto patrón.* El formato de una ecuación es:

```
--- f p <= t ;
```

Una ecuación consta de un lado izquierdo (o cabecera) y un lado derecho (o cuerpo). El lado izquierdo contiene el símbolo

de función **f** y un patrón **p**. El lado derecho contiene un término cualquiera **t**.

Función recursiva: Es una función tal que su declaración incluye su símbolo en el lado derecho de alguna ecuación. Dicha ecuación se llama una ecuación de recurrencia.

Función completa: El comportamiento completo de una función se define a partir del conjunto de sus ecuaciones. Cada ecuación puede usar cualquier combinación de variables y constructores en su patrón siempre que se respeten las siguientes restricciones:

- El conjunto de las ecuaciones debe cubrir con sus patrones el total de formas de construir el parámetro de la función. Si la función es parcial, deben faltar las ecuaciones correspondientes a los casos indefinidos.
- Dada una aplicación de la función a un valor constante, no debe existir ambigüedad sobre la ecuación a utilizar.

Variables anónimas y sinónimas: Una variable anónima de una ecuación es una variable del lado izquierdo que no aparece en el lado derecho de la ecuación. Una variable anónima no se representa en la cabecera con un símbolo de variable, sino con el carácter de subrayado '_'.

En una ecuación una variable sinónima **s** es una variable que representa a un subtérmino **t** del patrón **p** de la ecuación. Una variable sinónima se declara en el lado izquierdo de la ecuación escribiendo el patrón **p[t]** como **p[s&t]**.

Función selectora: Es una función que extrae un elemento (constante) de un valor formado por varios elementos.

Ambito de un símbolo: Es la porción de texto de programa donde el símbolo puede aparecer. Un símbolo de constante o función tiene como ámbito todo el texto que hay a partir de la definición del símbolo.

Un símbolo de una variable **x** que aparece en el lado izquierdo de una ecuación tiene como ámbito el lado derecho de dicha ecuación. Un símbolo de variable **x** que aparece en el término cualificado de una definición local tiene como ámbito el término resultado de la definición.

APENDICE B. TEORIAS DE LOS EJEMPLOS

Incluimos la teoría combinada necesaria para desarrollar los ejemplos desarrollados por el autor durante la realización de la tesis. La relación completa de éstos se encuentra al comienzo del Capítulo 4, aunque con detalle sólo se hayan desarrollado cinco programas.

Se incluyen teorías de pares, listas y conjuntos de enteros. Por brevedad no se incluyen las instancias correspondientes de los principios de sustitutividad por igualdad. Las teorías de otros pares (entero-lista, lista-entero, etc.), n-tuplas, listas y conjuntos se definen de manera análoga, pero de nuevo no aparecen por brevedad.

B.1. TEORIA DE ENTEROS NO NEGATIVOS

Unicidad:

(∀ x:entero) not succ x=0 (unicidad-cero)

(∀ x,y:entero) if succ x=succ y then x=y (unicidad-succ)

Igualdad e inducción:

(∀ x:entero) x=x (=-reflexividad)

(∀ x,y:entero) if x=y then y=x (=-simetría)

(∀ x,y,z:entero)
if x=y and y=z then x=z (=-transitividad)

$$(\forall\ *)\ \left[\begin{array}{l}\text{if } \left[\begin{array}{l}F[0]\\ \quad\text{and}\\ (\forall\ x\text{:entero})\ \left[\begin{array}{l}\text{if } F[x]\\ \text{then } F[x+1]\end{array}\right]\end{array}\right]\\ \text{then } (\forall\ x\text{:entero})\ F[x]\end{array}\right] \qquad \text{(inducción)}$$

Descomposición:

$$(\forall\ x\text{:entero})\ \left[\begin{array}{c}x=0\\ \text{or}\\ (\exists\ y\text{:entero})\ x=\text{succ } y\end{array}\right] \qquad \text{(descomposición)}$$

Relación $<_{pred}$:

(∀ x,y:entero) $x <_{pred} y \equiv$ succ x = y ($<_{pred}$)

Suma:

(∀ x:entero) x+0 = x (+-cero)

(∀ x,y:entero) x+succ y = succ(x+y) (+-succ)

Resta:

(∀ x:entero) x-0 = x (resta-cero)

(∀ x,y:entero) (x+1)-(y+1) = x-y (resta-succ)

Multiplicación:

(∀ x:entero) x*0 = 0 (*-cero)

(∀ x,y:entero) x*succ y = x*y+x (*-succ)

Relación ≤:

(∀ x:entero) x≤0 ≡ x=0 (≤-cero)

(∀ x,y:entero) x≤y ≡ x=y or x≤y-1 (≤-predecesor)

(∀ x:entero) x≤x (≤-reflexividad)

(∀ x,y:entero) if x≤y and y≤x then x=y (≤-antisimetría)

(∀ x,y,z:entero)
if x≤y and y≤z then x≤z (≤-transitividad)

(∀ x,y:entero) x≤y or y≤x (≤-totalidad)

Relaciones asociadas a ≤:

(∀ x,y:entero) x<y ≡ x≤y and not x=y (menor-que)

(∀ x,y:entero) x≤y ≡ x<y or x=y (<-cierre-reflexivo)

(∀ x,y:entero) x≥y ≡ y≤x (mayor-o-igual-que)

(∀ x,y:entero) x≥y or y≥x (≥-totalidad)

(∀ x,y:entero) x>y ≡ y<x (mayor-que)

(∀ x,y:entero) if x>y then x≥y (>-≥-subrelación)

(∀ x:entero) succ x > 0 (>-cero)

(∀ x,y:entero) x>0 and y≥x ≡ y-x<y (<-resta)

Relación divisor:

(∀ x:entero) x|x (|-reflexiva)

(∀ x:entero) x|0 (|-cero)

$$(\forall\, x,y,z{:}\text{entero})\ \left[\ \text{if } y\geq x\ \text{ then }\left[\ z|x \text{ and } z|y \equiv z|x \text{ and } z|(y-x)\ \right]\right] \qquad (|\text{-resta})$$

$$(\forall\, x,y{:}\text{entero})\ \ \text{if } x|y \text{ and } y>0 \text{ then } x\leq y \qquad (|\text{-comparación})$$

$$(\forall\, x,y,z{:}\text{entero})\ \ z|x \text{ and } z|y \equiv z|x \text{ and } z|(y \bmod x) \qquad (|\text{-mod})$$

B.2. TEORIA DE PARES DE ENTEROS

Descomposición y unicidad:

$$(\forall\, x{:}\text{entero}\#\text{entero})\ \ x = (x1,x2) \qquad \text{(descomposición)}$$
$$(\exists\, x1,x2{:}\text{entero})$$

$$(\forall\, x1,x2,y1,y2{:}\text{entero})\ \left[\ \text{if } (x1,x2)=(y1,y2)\ \text{ then } x1=y1 \text{ and } x2=y2\ \right] \qquad (\text{unicidad },)$$

Igualdad:

$$(\forall\, p{:}\text{entero}\#\text{entero})\ \ p=p \qquad (=\text{-reflexiva})$$

$$(\forall\, p1,p2{:}\text{entero}\#\text{entero})\ \ \text{if } p1=p2 \text{ then } p2=p1 \qquad (=\text{-simetría})$$

$$(\forall\, p1,p2,p3{:}\text{entero}\#\text{entero})\ \ \text{if } p1=p2 \text{ and } p2=p3 \text{ then } p1=p3 \qquad (=\text{-transitividad})$$

Funciones selectoras:

$$(\forall\, x1,x2{:}\text{entero})\ \ \text{primero}\ (x1,x2) = x1 \qquad \text{(primero)}$$

$$(\forall\, x1,x2{:}\text{entero})\ \ \text{segundo}\ (x1,x2) = x2 \qquad \text{(segundo)}$$

Relaciones lexicográficas:

$$(\forall\, x1,x2,x3,x4{:}\text{entero})\ \left[\ (x1,x2) <_{\text{lex}(\leq_1,<_2)} (x3,x4) \equiv x1 <_1 x3 \text{ or } x1=x3 \text{ and } x2 <_2 x4\ \right] \qquad (<_{\text{lex}(<1,<2)})$$

para dos relaciones $<_1$ y $<_2$ cualesquiera

$$(\forall\, x1,x2,x3,x4{:}\text{entero})\ \left[\ (x1,x2) <_{\text{prog}(\leq_1,<_2)} (x3,x4) \equiv x1 <_1 x3 \text{ and } x2=x4 \text{ or } x1=x3 \text{ and } x2 <_2 x4\ \right] \qquad (<_{\text{prog}(<1,<2)})$$

para dos relaciones $<_1$ y $<_2$ cualesquiera

B.3. TEORIA DE LISTAS DE ENTEROS

Unicidad:

(∀ x:entero,l:lista) not x::l=nil (unicidad-nil)

(∀ x1,x2:entero) (∀ l1,l2:lista) [if x1::l1=x2::l2 then x1=x2 and l1=l2] (unicidad-::)

Igualdad e inducción:

(∀ l:lista) l=l (=-reflexiva)

(∀ l1,l2:lista) if l1=l2 then l2=l1 (=-simetría)

(∀ l1,l2,l3:entero) if l1=l2 and l2=l3 then l1=l3 (=-transitividad)

$$(\forall\ *)\ \left[\begin{array}{l}\text{if } \left[\begin{array}{l}F[nil]\\ \quad\text{and}\\ (\forall\ x{:}entero)\\ (\forall\ l{:}lista)\end{array}\right.\ \left[\begin{array}{l}\text{if } F[l]\\ \text{then } F[x::l]\end{array}\right]\\ \text{then } (\forall\ l{:}lista)\ F[l]\end{array}\right] \quad \text{(inducción)}$$

para cualquier sentencia F[l] donde x no aparezca libre

Descomposición:

$$(\forall\ l{:}lista)\ \left[\begin{array}{l} l=nil\\ \quad or\\ (\exists\ x{:}entero;l'{:}lista)\ l=x::l'\end{array}\right] \quad \text{(descomposición)}$$

Funciones selectoras:

(∀ x:numero;l:lista) cabeza (x::l) = x (cabeza)

(∀ x:numero;l:lista) cola (x::l) = l (cola)

Relación $<_{cola}$:

$$(\forall\ l1,l2{:}lista)\ \left[\begin{array}{c} l1 <_{cola} l2\\ \equiv\\ (\exists\ x{:}entero)\ x::l1 = l2\end{array}\right] \quad (<_{cola})$$

(∀ l:lista) not l $<_{cola}$ nil ($<_{cola}$-nil)

(∀ x:entero;l:lista) l $<_{cola}$ x::l ($<_{cola}$-::)

Funciones de partición:

(∀ x,y:entero;l:lista) izda(x::y::l)<>dcha(x::y::l)=x::y::l (izda-dcha)

(∀ x,y:entero;l:lista) not izda(x::y::l)=x::y::l (izda)

(∀ x,y:entero;l:lista) not dcha(x::y::l)=x::y::l (dcha)

Concatenación:

(∀ l:lista) nil<>l = l (<>-nil-izdo)

(∀ x:entero;l1,l2:lista)
(x::l1)<>l2 = x::(l1<>l2) (<>-::)

(∀ l:lista) l<>nil = l (<>-nil-dcho)

(∀ x:entero;l:lista) x::nil<>l = x::l (<>-átomo-izdo)

(∀ l1,l2,l3:lista)
(l1<>l2)<>l3 = l1<>(l2<>l3) (<>-asociativa)

Inversión:

invertir nil = nil (invertir-nil)

(∀ x:entero;l:lista)
invertir (x::l) = invertir l<>x::nil (invertir-::)

Relación de pertenencia:

(∀ x:entero) not x∈nil (∈-nil)

(∀ x,y:entero;l:lista) x∈y::l ≡ x=y or x∈l (∈-::)

(∀ x:entero;l1,l2:lista) x∈l1<>l2 ≡ x∈l1 or x∈l2 (∈-<>)

Relación de permutación:

(∀ l:lista) perm (l,l) (perm-reflexiva)

(∀ x:entero;l1,l2:lista)
perm (x::l1,x::l2) ≡ perm (l1,l2) (perm-::)

(∀ x:entero;l1,l2,l3:lista)
perm (x::l1,l2<>x::l3) ≡ perm (l1,l2<>l3) (perm-<>)

(∀ l1,l2:lista) perm(l1<>l2,l2<>l1) (perm-<>-conmutativa)

Predicado de ordenación:

ord nil (ord-nil)

ord x::nil (ord-átomo)

(∀ x:entero;l:lista)

$$\text{ord } (x::l) \equiv \left[\begin{array}{c}\text{ord } l \\ \text{and} \\ (\forall\ y\text{:entero})\ \text{if } y\in l \text{ then } x\leq y\end{array}\right] \quad \text{(ord-::)}$$

$$(\forall\ l1,l2:\text{lista})\ \left[\ \text{ord}\ (l1<>l2) \equiv \left[\ \text{ord}\ l1\ \text{and}\ \text{ord}\ l2\ \text{and}\ (\forall\ x,y:\text{entero})\ \left[\ \text{if}\ x\epsilon l1\ \text{and}\ y\epsilon l2\ \text{then}\ x\leq y\ \right]\right]\right] \qquad (\text{ord-}<>)$$

Otras relaciones:

$$(\forall\ x:\text{entero})\ (\forall\ l1,l2,l3:\text{lista})\ \left[\ \text{if perm}(l1<>x::l2,l3)\ \text{then}\ l2<_{saco}l3\ \right] \qquad (<_{saco})$$

$$(\forall\ l1,l2,l3,l4:\text{lista})\ \left[\ \text{if}\ l1=l2<>l3<>l4\ \text{and not}\ l1=l3\ \text{then}\ l3<_{lista}l1\ \right] \qquad (<_{lista})$$

B.4. TEORIA DE CONJUNTOS DE ENTEROS

Relación de pertenencia:

$$(\forall\ x:\text{entero})\ \text{not}\ x\epsilon\phi \qquad (\epsilon\text{-}\phi)$$

$$(\forall\ x,y:\text{entero};c:\text{conjunto})\ x\epsilon y.c \equiv x=y\ \text{or}\ x\epsilon c \qquad (\epsilon\text{-}.)$$

$$(\forall\ x:\text{entero};c1,c2:\text{conjunto})\ x\epsilon c1Uc2 \equiv x\epsilon c1\ \text{or}\ x\epsilon c2 \qquad (\epsilon\text{-}U)$$

Igualdad e inducción:

$$(\forall\ c:\text{conjunto})\ c=c \qquad (=\text{-reflexiva})$$

$$(\forall\ c1,c2:\text{conjunto})\ \text{if}\ c1=c2\ \text{then}\ c2=c1 \qquad (=\text{-simetría})$$

$$(\forall\ c1,c2,c3:\text{conjunto})\ \text{if}\ c1=c2\ \text{and}\ c2=c3\ \text{then}\ c1=c3 \qquad (=\text{-transitividad})$$

$$(\forall\ x:\text{entero};c:\text{conjunto})\ x.(x.c) = x.c \qquad (\text{multiplicidad})$$

$$(\forall\ x,y:\text{entero};c:\text{conjunto})\ x.(y.c) = y.(x.c) \qquad (\text{intercambio})$$

$$(\forall^*)\ \left[\ \text{if}\ \left[\ F[\phi]\ \text{and}\ (\forall\ x:\text{entero})\ (\forall\ c:\text{conjunto})\ \left[\text{if not}\ x\epsilon c\ \text{then if}\ F[c]\ \text{then}\ F[x.c]\right]\right]\ \text{then}\ (\forall\ c:\text{conjunto})\ F[c]\ \right] \qquad (\text{inducción})$$

para cualquier sentencia F[c] donde x no aparezca libre

Descomposición:

$$(\forall\ c{:}\text{conjunto})\quad \left[\begin{array}{l} c=\phi \\ \text{or} \\ (\exists\ x{:}\text{entero}\ (\exists\ c'{:}\text{conjunto})\ [\ c=x.c'\ \text{and}\ \text{not}\ x\epsilon c'\ ]\end{array}\right] \qquad \text{(descomposición)}$$

Relación $<_{resto}$:

$$(\forall\ x{:}\text{entero}; c{:}\text{conjunto})\quad \text{if not } x\epsilon c \text{ then } c <_{resto} x.c \qquad (<_{resto})$$